Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr. Dr. Theo Härder

Datenbank-Caching:

Modellierung des Füllverhaltens von Cache-Groups

Diplomarbeit

von

Christian Bayerlein

Oktober 2004

Betreuer:
Andreas Bühmann

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

Koblenz, 30. Oktober 2004

Christian Bayerlein

Zusammenfassung

Der Einsatz von Caching-Technologien die in weiten Bereichen eingesetzt werden, ist sehr vorteilhaft, wenn es darum geht, Daten repliziert zwischenzuspeichern, weil auf die Originaldaten nur kostenintensiv (im Sinne der Antwortzeit des Servers) zugegriffen werden kann. Bekannte Beispiele sind Prozessor-Caches, die eine kleine Teilmenge der gesamten Daten aus dem Hauptspeicher lokal (d. h. nahe beim Prozessor) zur Verarbeitung vorhalten oder der Datenbank-Puffer, der Seiten im Hauptspeicher vorhält, damit nicht bei jeder Anfrage auf den Sekundärspeicher (z. B. die Festplatte) zugegriffen werden muss. Auch in Netzwerk-Umgebungen spielen Caching-Technologien in letzter Zeit immer häufiger eine wichtige Rolle. Web-Caching ermöglicht das Speichern von (statischen) Web-Objekten, wie z. B. HTML- oder XML-Fragmenten auf dem Pfad zwischen Backend-Server und dem Benutzer-Agenten (z. B. einem Web-Browser). Im Gegensatz dazu bietet Datenbank-Caching einen Ansatz zur Zwischenspeicherung von dynamischen Daten mit Hilfe von vollständigen Datenbank-Management-Systemen. Mit Hilfe von Cache-Groups für Gleichheitsprädikate, einem Spezialfall des Constraint-basierten Datenbank-Caching kann der Cache durch ein adaptives Verhalten dynamisch auf sich ändernde Anfragecharakteristika optimiert werden. Die zu speichernden Objekte stellen dabei Extensionen parametrisierter Prädikate dar, und das Verhalten des Caches wird über Constraints gesteuert. Cache-Groups, die für praktische Anwendungen eingesetzt werden, müssen ein kontrollierbares Füllverhalten aufweisen, denn ansonsten kommt es schnell durch ein exzessives Füllverhalten dazu, dass komplette Tabellen aus dem Backend im Frontend repliziert gespeichert werden. Dies macht aber wiederum auf Grund der hohen Replikations- und Wartungskosten die Vorteile des Caching zunichte. Diese Arbeit stellt Modelle zur quantitativen Analyse des Füllverhaltens von Cache-Groups vor. Dabei werden drei verschiedene Arten von Cache-Groups untersucht und jeweils die Anzahl der in den unterschiedlichen Tabellen im Cache (Frontend) gespeicherten Sätze abgeschätzt, denn diese Größe spielt eine zentrale Rolle bei der Bewertung der Kosten zur Verwaltung einer Cache-Group.

Inhaltsverzeichnis

Zusammenfassung **5**

Inhaltsverzeichnis **7**

1 Einleitung **9**
1.1 Web-Caching 10
1.2 Datenbank-Caching 12
1.3 Constraint-basiertes Caching 14
1.4 Gleichheitsprädikate 16
1.5 Cache-Groups für Gleichheitsprädikate 19
1.6 Abgrenzung zu anderen Arten des DB-Caching 22
1.7 Anfrageverarbeitung und Füllverhalten 24
1.8 Quantitative Analyse 28

2 Grundlagen **33**
2.1 Modellannahmen 34
2.2 Der Spalten-Beziehungsgraph 35
2.3 Füllschritte 37
2.4 Werte und Wertemengen 41
2.5 Exkurs: Abschätzung der Schnittmengenmächtigkeit 45
2.6 Histogramme und Zugehörigkeitsfunktion 48
2.7 Verhalten von Cache-Keys 49
2.8 RCCs 51
2.8.1 Die RCC-Eigenschaft 51
2.8.2 Die RCC-Teilmengen-Annahme 54
2.8.3 Überdeckung zwischen zwei Spalten 56
2.8.4 Anzahl der über eine RCC induzierte Werte 57
2.8.5 Wertemengen-Pfade 58
2.9 Schmuggler-Beziehungen 63
2.10 Zusammenfassung 68

3 Baumartige Cache-Groups **73**
3.1 Grundlegende Zusammenhänge und Eigenschaften 74
3.2 Modellierung eines Füllzyklus 76
3.3 Verhalten von Cache-Tabellen ohne Cache-Key 77
3.4 Algorithmus zur Abschätzung der Füllstände 78
3.4.1 Abschätzung der Füllstände ohne Histogramme 78
3.4.2 Betrachtung des Algorithmus anhand eines Beispiels 80
3.5 Algorithmus mit Histogrammen 83
3.5.1 Zusammenfassung der maßgeblichen Berechnungsvorschriften 84
3.5.2 Erster Entwurf 85
3.5.3 Steigerung der Effizienz 87

4 Nicht-baumartige, RCC-zyklenfreie Cache-Groups ... 90
4.1 Der SBG nicht-baumartiger, RCC-zyklenfreier Cache-Groups ... 90
4.2 Abschätzung der Füllstände ... 91
4.3 Schwachpunkte des Algorithmus ... 95

5 Allgemeine Cache-Groups und -Föderationen ... 99
5.1 Token und Stellen ... 99
5.2 Semantik der Token ... 103
5.2.1 Besuche und Historien ... 103
5.2.2 Informationen aus der Historie ... 106
5.2.3 Besuchsbäume ... 107
5.3 Kontexte und Kontext-Bäume ... 109
5.4 Abschätzung der Anzahl zusätzlich geladener Werte ... 112
5.4.1 Anzahl generell zu ladender Werte ... 113
5.4.2 Anzahl der zusätzlich geladenen Werte ... 114
5.5 Veränderung des Füllstands ... 116
5.6 Aktualisierung der Statistik im Konsumenten ... 117
5.7 Erzeugung neuer Token und Schmuggeln von Werten ... 118
5.8 Überblick über einen Füllzyklus ... 120

6 Evaluierung ... 123
6.1 Population von Backend-Tabellen ... 124
6.2 Ablauf der Evaluation, Implementierung des Verfahrens ... 125
6.3 Beispielhafte Evaluierung ... 127

7 Zusammenfassung und Ausblick ... 131
7.1 Grundlagenteil ... 131
7.2 Hauptteil ... 133
7.2.1 Baumartige Cache-Groups ... 133
7.2.2 Nicht-baumartige, RCC-zyklenfreie Cache-Groups ... 135
7.2.3 Allgemeine Cache-Groups und -Föderationen ... 136
7.3 Evaluierung ... 138
7.4 Ausblick ... 138

Anhang A: Literaturverzeichnis ... 141

Anhang B: Wichtige Schreibweisen und Symbolverzeichnis ... 143

Anhang C: Beweise ... 151

1 Einleitung

Webbasierte Anwendungen haben in letzter Zeit einen hohen Grad an Verbreitung und Benutzung in den verschiedensten Bereichen gewonnen. Sie werden unter anderem in den Gebieten e-Business, e-Government und e-Learning eingesetzt. Ihnen allen zugrunde liegt die Kommunikation mit Benutzern mit Hilfe von Internet-Technologien wie HTTP als Protokoll und (X)HTML zur Formatierung der Ausgabe. Dabei spielt die Interaktion zwischen Benutzer und Anwendung eine wichtige Rolle, wobei die Ausgabe dynamisch bereitgestellt wird.

Anwendungen in diesem Bereich sind in den letzten Jahren immens an Umfang und Anzahl gewachsen. Dieses Wachstum scheint auch in näherer Zukunft ungebrochen zu sein. Neuere Entwicklungen auf dem Bereich der Web-Services [W3C04] (d. h. dem Datenaustausch zwischen Anwendungen über Netzwerke mit Hilfe von XML-Dokumenten) tragen sicherlich in Zukunft zu einer noch weiteren Verbreitung der e-*-Anwendungen bei, da eine Integration einer Web-Oberfläche (mit (X)HTML) nahe liegt. Zudem sind Anwendungen im Bereich der Web-Services bezüglich ihrer Architektur sehr ähnlich zu „klassischen“ webbasierten Anwendungen.

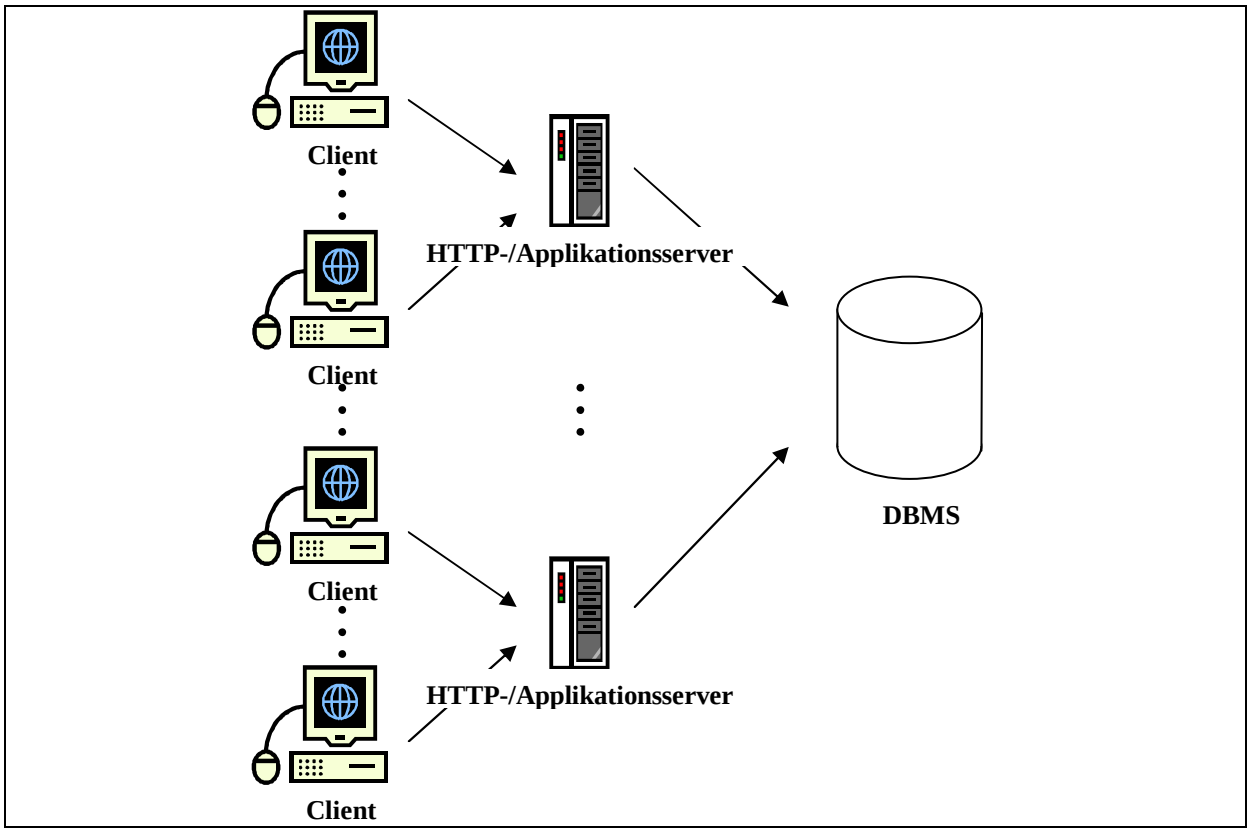

Abbildung 1: Eine typische Mehrschichtarchitektur

Die Umsetzung solcher Applikationen geschieht üblicherweise unter Zuhilfenahme einer Vielzahl von Technologien. Darunter fallen unter anderem automatische Netzwerk-Lastverteilung, HTTP-Server, Applikationsserver und (objekt-)relationale Datenbank-Management-Systeme (DBMS).

In ihrer einfachsten Form wird eine webbasierte Anwendung lediglich durch einen Applikationsserver mit HTTP-Schnittstelle realisiert, der Darstellungslogik und Anwendungslogik zur Verfügung stellt. Die Darstellungslogik kann dabei z. B. mit Hilfe von Java-Server-Pages (X)HTML-Seiten dynamisch generieren und die Anwendungslogik kann mit Java-Servlets oder Enterprise-Java-Beans realisiert sein. Dabei greift die Anwendungslogik wiederum auf Daten zu, die in einem DBMS gespeichert sind. Eine derartig aufgebaute Infrastruktur wird auch n-Tier-Architektur bzw. Mehrschichtarchitektur genannt. Abbildung 1 zeigt eine solche typische Architektur.

Durch die vielen verschiedenen Schichten in einer Mehrschichtarchitektur kommt es oftmals zu einer Verschlechterung der Antwortzeit und der Skalierbarkeit der realisierten Webanwendung. Die Begründung dafür ist darin zu sehen, dass die benötigten Objekte im schlechtesten Fall jedes Mal über den gesamten Pfad zwischen Backend-DBMS und Browser transportiert werden müssen. Schließlich müssen die Komponenten der Anwendung (also HTTP-, Applikations- und DB-Server sowie weitere Teile) nicht auf der selben Maschine laufen, sondern können auch über ein Netzwerk miteinander verbunden sein. Hierdurch spielt natürlich die (räumliche) Entfernung eine Rolle bei der Antwortzeit. Das Netzwerk zwischen den Komponenten stellt oft einen Flaschenhals dar.

Eine übliche Interaktion eines Benutzers mit der Anwendung läuft so ab, dass der Benutzer eine Anfrage an die Webanwendung stellt, indem er mit seinem Browser eine der Seiten der Applikation aufruft. Der Browser schickt daraufhin eine Anforderung an den HTTP-Server, der wiederum die Anfrage analysiert und entsprechend aufbereitet an den Applikationsserver weiterleitet. Dieser bearbeitet die Anfrage mit Hilfe der in ihm implementieren Anwendungslogik. Dabei holt er die zur Beantwortung der Anfrage benötigten Objekte aus dem Backend-DBMS. Der Applikationsserver erzeugt aus den gelieferten Objekten mit Hilfe der Darstellungslogik eine Antwortseite (eventuell werden dabei statische HTML-Fragmente benutzt). Diese Antwortseite wird dann über den HTTP-Server zurück an den Browser des Benutzers geschickt.

1.1 Web-Caching

Ohne weitere Optimierung muss für jede Benutzeranfrage die Antwortseite neu generiert und dabei jedes Mal auf die Objekte im Backend-DBMS zugegriffen werden. Selbst statische Elemente wie Bilder, HTML-Fragmente usw. werden für jede Anfrage mindestens vom Applikationsserver (der daraus zusammen mit dynamischen Elementen das Antwort-Dokument erstellt) über den HTTP-Server zum Browser des Benutzers geschickt. Aufgrund der steigenden Zahl der Anfragen pro Zeiteinheit (verursacht durch eine größer werdende Anzahl an Benutzern) würde der Applikationsserver schnell überlastet und käme bezüglich Antwortzeit und Verfügbarkeit an die Grenzen des (für die Benutzer) Erträglichen. Aus diesem Grund gibt es in letzter Zeit eine Reihe von Ansätzen, um eine bessere Skalierbarkeit

und Antwortzeit zu erzielen, die auf das Zwischenspeichern, d. h. auf ein lokales Vorhalten, von Daten, aufbauen.

Diese Techniken werden als Caching-Verfahren bezeichnet. Der englische Begriff Cache bedeutet wörtlich übersetzt geheimes Lager oder Versteck, wird aber in der Informatik als Zwischenspeicher übersetzt. In [Wiki04b] finden wir folgende Definition zum Begriff Cache:

> A **cache** is a collection of duplicate data, where the original data is expensive to fetch or compute (usually in terms of access time) relative to the cache. Future accesses to the data can be made by accessing the cached copy rather than refetching or recomputing the original data, so that the perceived average access time is lower.

Frei übersetzt bedeutet diese Definition: Ein Cache ist eine Sammlung duplizierter Daten, bei denen die Originaldaten im Vergleich zum Cache nur kostenintensiv (meistens im Sinn von Antwortzeit) zu holen oder berechnen sind. Zukünftige Zugriffe auf die Daten können dann durch Verwendung der zwischengespeicherten Kopien statt erneutem Holen oder Berechnen der Originaldaten gemacht werden, so dass die erreichte mittlere Zugriffszeit verringert wird.

Der Grund, warum Caching überhaupt effektiv funktioniert, ist, dass viele Zugriffsmuster in üblichen Anwendungen eine so genannte Referenzlokalität aufweisen. Diese Referenzlokalität existiert in vielerlei Ausprägungen, aber üblicherweise ist damit gemeint, dass die gleichen Daten häufig nacheinander oder in kurzer zeitlicher Abfolge abgerufen werden oder dass nahe beieinander gespeicherte Daten kurz hintereinander abgerufen werden. Aus diesem Grund kann ein Zwischenspeichern von kürzlich verwendeten Daten in einem Cache sinnvoll sein, da in der Regel mit einer baldigen erneuten Verwendung dieser Daten zu rechnen ist. Die Daten können dann direkt aus dem Cache geliefert werden, ohne auf den eigentlichen Speicherplatz zugreifen zu müssen.

Beim Caching innerhalb von webbasierten Anwendungen geht es um das Zwischenspeichern von benutzten Web-Objekten in einem Cache. Das Zwischenspeichern (Caching) beschränkte sich dabei bisher auf statische HTML-Seiten, XML-Dokumente und Fragmente als duplizierte Objekte und fand auf verschiedenen Ebenen statt [HB04a]:

- Im Browser bzw. Benutzeragenten,
- mit Hilfe eines (HTTP-)Proxies,
- in Knoten eines CDNs (Content-Delivery-Overlay-Networks) [1] oder
- in speziellen Objekt-Caches für statische Objekte als Teil der Anwendungslogik.

Dieses so genannte **Web-Caching** bietet für die bei webbasierten Anwendungen auftretenden Lastcharakteristika bei statischen Web-Objekten die allgemeinen Vorteile von Caching [HB04a]:

- Entlastung der Kommunikationsverbindungen und Reduktion der benötigten Übertragungsbandbreiten,
- Verkürzung der Antwortzeit für die Aufrufe durch Web-Benutzer und

[1] Akamai betreibt solche Netze, die heute bereits 15000 Knoten (Edge Caching Servers) umfassen. [ABK+03]

- Entlastung des (eigentlich aufgerufenen) Web- bzw. Applikationsservers.

Ebenfalls durch den Einsatz von statischem Caching ergibt sich aus Sicht des Benutzers eine bessere Verfügbarkeit zumindest der statischen Teile der Web-Applikation.

Einsatz und Nutzung dieser Web-Caches wurden deshalb in den letzten Jahren stark ausgedehnt und optimiert; über die Vielfalt der Ansätze sowie ihrer Optimierungs- und Ersetzungsstrategien berichtet [PB03].

Durch Web-Caching kann allerdings nicht die Problematik gelöst werden, dass auf dynamisch veränderbare Objekte in einer Backend-Datenbank für jede Anfrage neu zugegriffen werden muss, wenn diese z. B. aktuell sein müssen. Gerade diese Problematik gewinnt in letzter Zeit aber immer mehr an Bedeutung. Durch die stärker werdende Personalisierung und die wachsende Erfordernis, sich häufig verändernde aktuelle Daten zu liefern, werden die webbasierte Anwendungen und die Ihnen zugrunde liegenden Daten zunehmend dynamisch. Große Websites liefern zunehmend persönlich zusammengestellte (z. B. mit individueller Werbung oder einem persönlichen Warenkorb), dynamisch generierte Inhalte an ihre Benutzer. Wir benötigen entsprechend der Dynamik der Daten neue Technologien, um veränderbare Objekte aus einer Datenbank zwischenspeichern zu können.

Datenbank-Caching (DB-Caching) ist eine viel versprechende neuartige Technik, welche sich der Problematik, die sich durch die Dynamik der Objekte in webbasierten Anwendungen ergibt, widmet. Die Anwendung greift auf die Daten, welche in einem Datenbank-Cache zwischengespeichert sind, mit Hilfe von Anfragen zu; also auf genau die gleiche Art, wie sie auf die Backend-Datenbank zugreifen würde. Tatsächlich weiß in vielen Fällen die Applikation nichts über die Existenz des Datenbank-Caching. Das Caching läuft also für die Anwendung transparent ab.

1.2 Datenbank-Caching

Beim DB-Caching geht es um das (duplizierte) Zwischenspeichern von Objekten einer Datenbank aus einer **Backend-DB** (bzw. kurz **Backend**) in einem lokalen Cache (bzw. **Frontend**). Ziel des DB-Caching ist es, zukünftige Anfragen aus dem Cache beantworten zu können. Anwendungen, welche die Daten aus der Datenbank benutzen, sollten dabei nicht verändert werden müssen, damit der Cache benutzt werden kann. Werden die Anfragen aus dem Cache beantwortet, so müssen die aus dem Cache gelieferte Antwort korrekt sein, d. h. die Antwort muss mit der Antwort übereinstimmen, wie sie vom Backend auf die gleiche Anfrage geliefert worden wäre (bis auf die abgeschwächte Anforderungen bezüglich der Aktualität der Daten, falls dies akzeptiert werden kann).

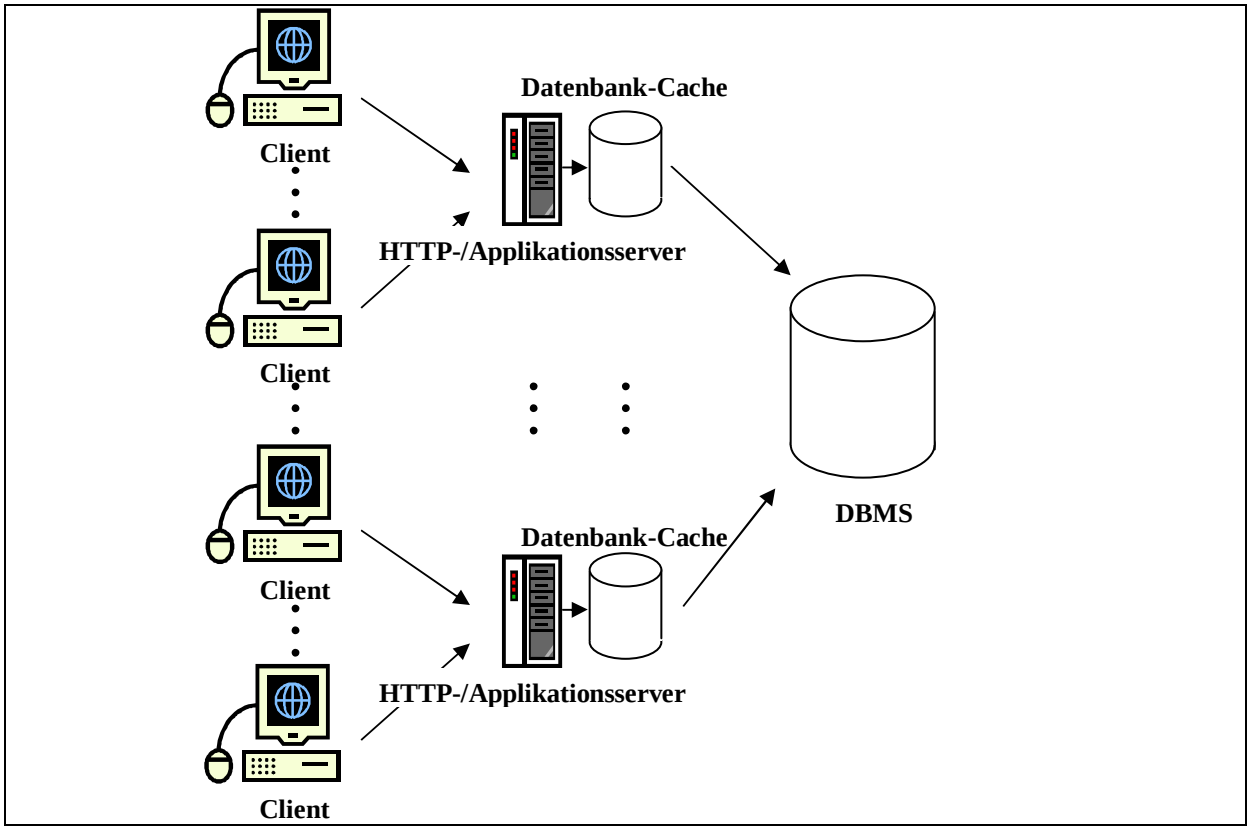

Abbildung 2: Eine Mehrschichtarchitektur mit Datenbank-Caches

In Abbildung 2 ist dargestellt, wie eine übliche Mehrschichtarchitektur um Datenbank-Caches erweitert werden kann. Die Caches halten jeweils eine Teilmenge der Daten aus der Backend-Datenbank nahe beim Applikationsserver vor.

Das Vorhaben, dynamische Objekte lokal in einem Cache vorzuhalten, gestaltet sich keineswegs trivial.

Die Objekte im Cache müssen aktuell sein und somit müssen mögliche, im Backend stattfindende Änderungen der Objekte zeitnah im Cache nachvollzogen werden. Im Allgemeinen wird eine maximale Zeitspanne angegeben, innerhalb der die Objekte im Cache aktualisiert sein müssen. Dadurch kann sichergestellt werden, dass die Daten im Cache niemals älter als die angegebene Zeitspanne sind.

Zu dieser Aktualitätseigenschaft kommen zusätzliche Integritätsbedingungen (entweder als Cache-Constraints oder fest definierte materialisierte Sichten), die garantieren, dass die ankommenden Anfragen korrekt aus dem Cache beantwortet werden können, falls die notwendigen Daten vorhanden sind. Bei einer Antwort aus dem Cache muss das Ergebnis immer mit dem übereinstimmen, welches von der Backend-DB geliefert worden wäre. Die Problematik ergibt sich wegen des üblichen Einsatzes von SQL und der daraus resultierenden mengenorientierten Verarbeitung im DBMS. Konkret bedeutet dies, dass stets gewährleistet sein muss, dass (Teil-)Anfragen im Datenbank-Cache abgewickelt werden können.

Der einfachste Ansatz, um ein DB-Caching-System zu realisieren, wäre, den gesamten Inhalt ausgewählter Tabellen der Backend-Datenbank (Backend-Tabellen) im Frontend zu replizieren. Wir nennen dieses Verfahren Volltabellen-Caching. Im Fall von Volltabellen-Caching

kann jede durch eine Anfrage referenzierte Tabelle im Cache zur Beantwortung benutzt werden, solange die Aktualität der Daten akzeptabel ist. Dies ist möglich, da die Antwortmenge möglicher Anfragen auf dieser Tabelle Projektionen und Selektionen auf die in der Tabelle gespeicherten Daten darstellen. Anfragen können somit aus dem Cache beantwortet werden, wenn die Anfrage lediglich aus einer Transformation der zugrunde liegenden Tabelle besteht und somit keine Joins enthält.

Die Einfachheit dieses Ansatzes sorgte unter anderem für eine große Beachtung einiger DB-Caching-Produkte, die mit Volltabellen-Caching arbeiten. Allerdings sind die Frontend-Systeme üblicherweise wesentlich leistungsschwächer als die Backend-Systeme, wodurch ein Volltabellen-Caching schwierig bis unmöglich wird. Sogar für ein leistungsfähiges Frontend-System kann durch große Tabellen ein Volltabellen-Caching auf Grund von erhöhten Kosten für Replikation und Instandhaltung unmöglich werden. Lediglich für selten aktualisierte Tabellen kann ein Volltabellen-Caching als effektive Lösung gesehen werden. Hier sind die Replikations- und Wartungskosten sehr gering, da nur selten Updates im Cache nachvollzogen werden müssen.

Eine Alternative zum Volltabellen-Caching ist Teiltabellen-Caching. Bei diesem Verfahren wird lediglich eine Teilmenge der Sätze (also transformiert durch Projektion und Selektion) ausgewählter Tabellen im Cache gespeichert. Teiltabellen-Cache kann eine effektive Alternative zum Volltabellen-Caching sein, indem nur die interessanten Teile der Backend-Tabelle im Cache vorgehalten werden (eben die Antwortmengen von zukünftig wahrscheinlich benutzten Anfragen auf diese Tabelle).

Die Technologie materialisierter Sichten ist eine der Möglichkeiten, Teiltabellen-Caching zu realisieren, obwohl sie ursprünglich für andere Zwecke entwickelt wurde. In gegenwärtigen Datenbank-Produkten speichern materialisierte Sichten bereits berechnete Anfrageergebnisse, welche später benutzt werden, um den Zugriff auf die Daten bei komplexen Anfragen zu beschleunigen und die Leistung des DBMS somit zu verbessern.

Gekoppelt mit Mechanismen föderierter DBMS können materialisierte Sichten dazu eingesetzt werden, eine im Vorfeld explizit deklarierte Teiltabelle einer Backend-Tabelle im Cache zu halten. Diese Teiltabelle entsteht dabei aus einer Projektion und Selektion auf der Backend-Tabelle. Durch materialisierte Sichten können sogar Anfragen über mehrere Tabellen hinweg (also durch Joins gebildete Anfragen) aus dem Cache beantwortet werden, wenn für diese Anfrage eine materialisierte Sicht vom Administrator eingerichtet wurde. Im System MTCache der Firma Microsoft wird eine solche Technologie eingesetzt [LGZ04]. Allerdings ist dann eine explizite Deklaration notwendig und das Caching kann somit nicht flexibel und adaptiv auf eine sich ändernde Anfragecharakteristik reagieren.

1.3 Constraint-basiertes Caching

Datenbank-Caching durch materialisierte Sichten ist relativ unflexibel, da die materialisierten Sichten explizit vom Administrator definiert werden müssen. Aus diesem Grund kann nicht dynamisch auf sich ändernde Anfragecharakteristik reagiert werden. Der Administrator muss selbst abschätzen, welche Daten vermutlich oft von der Web-Anwendung benutzt werden und die materialisierten Sichten entsprechend deklarieren.

Wir stellen in dieser Arbeit eine andere Möglichkeit des Datenbank-Caching vor. Es handelt sich dabei um das so genannte **Constraint-basierte Caching**. Bei diesem Ansatz wird eine adaptive Verwaltung der im Cache gespeicherten Daten angestrebt. Der Cache-Inhalt soll sich also nach den in der Vergangenheit nachgefragten Daten richten. Um diese Ziele zu erreichen, werden beim Constraint-basierten Caching Bedingungen (Constraints) definiert, die vom Cache-Manager durchgesetzt werden, um den Cache immer in einem gültigen Zustand zu halten. In einem gültigen Zustand wird garantiert, dass wir entscheiden können, ob sich die angefragten Daten im Cache befinden. Deswegen wird das Nachladen von Daten in den Cache ebenfalls durch solche Constraints gesteuert.

Beim Constraint-basierten Datenbank-Caching arbeiten wir prinzipiell mit parametrisierten Prädikaten.

Definition: Ein **Prädikat** beschreibt Eigenschaften, die von einer Satzmenge erfüllt werden.[2] Ein **parametrisiertes Prädikat** ist ein Schema für ein Prädikat, das Parameter enthält. Wird ein bestimmtes Argument als Parameter eingesetzt, so wird das Prädikat instanziiert.

Parametrisierte Prädikate (bzw. darauf basierende Anfragen) spielen noch in anderen Bereichen eine wichtige Rolle. So ist es z. B. in JDBC [Sun04a] (der DB-Schnittstelle von Java) möglich, Anfragen mit Hilfe so genannter „Prepared Statements“ [Sun04b] vorbereiten zu lassen. Die Datenbank kann dann im Voraus den Zugriffsplan optimieren und für die Beantwortung der anschließenden (mehrfachen) Instanziierungen der Anfrage wieder verwenden. Ein „Prepared Statement“ hat dabei die Form einer üblichen SQL-Anfrage, wobei die Parameter durch ein Fragezeichen eingebaut werden.

Die **Extension** eines (SQL-)Prädikats ist die Satzmenge, die als Grundlage für die Beantwortung einer Anfrage mit diesem Prädikat dient. Wenn die Anfrage keine weiteren Transformationen (z. B. Join oder Projektion) enthält, so entspricht die Extension der Antwortmenge des der Anfrage zugehörigen Prädikats.

Beispiel: Wir betrachten die in Tabelle 1 abgebildete Relation. Die SQL-Abfrage

```
SELECT * FROM BENUTZER WHERE Nachname='Bayerlein'
```

würde die durch einen Haken ✓ markierten Sätze als Antwort bekommen. Bei dieser Satzmenge handelt es sich um die Extension unserer SQL-Abfrage.

[2] Spätestens seit Aristoteles wird in der logischen Analyse von einfachen (kategorischen) Aussagen und Urteilen unterschieden zwischen dem, von dem etwas gesagt wird (der Gegenstand, das Subjekt), und dem, das von etwas gesagt wird (das Prädikat). [Wiki04c]

	Vorname	Nachname
✓	Christian	Bayerlein
	Cornelia	Weber
✓	Michael	Bayerlein
	Gabi	Hofmann
	Gudrun	Wagner
	Sara	Kreutz
	Andreas	Bühmann
	Silke	Duck

Tabelle 1: Eine Relation „EHRENVOLLE_HELFER“ als Beispiel für die Extension eines Prädikats

Wir können die Anfrage parametrisieren, indem wir den Wert „Bayerlein“ durch einen Parameter ersetzen. Durch Einsetzen eines Werts (z. B. „Bayerlein“, aber auch „Weber“ oder „Hofmann“ würde das Schema instanziiert und wir könnten die Extension des instanziierten Prädikats betrachten.

Man beachte, dass dieses Beispiel sehr einfach gewählt ist. Eine Extension eines Prädikats kann durchaus durch Satzmengen aus verschiedenen Tabellen gebildet sein. Solche Satzmengen werden im Allgemeinen über Verbundanfragen (zum Beispiel mit Equi-Join-Prädikaten) angefordert. Die Satzmengen in den am Verbund beteiligten Tabellen tragen dann zur Beantwortung der Anfrage bei und erfüllen die auf ihre Tabelle bezogene, durch das Prädikat beschriebene Eigenschaft (z. B. die Existenz eines Verbundpartners in einer anderen Tabelle).

Die Objekte in unserem Cache sind also Extensionen (instanziierter) parametrisierter Prädikate. Die Instanzen der Prädikate wurden dabei in jüngster Vergangenheit in Anfragen verwendet. Beim allgemeinen Constraint-basierten Caching richtet sich die Art der unterstützten Prädikate nach den Constraints, die im Cache definiert werden können.

1.4 Gleichheitsprädikate

Wie wir gesehen haben, unterstützt das Constraint-basierte DB-Caching die Beantwortung von Anfragen durch ein Vorhalten der Extension bestimmter Prädikate im Cache. Damit wir prüfen können, ob eine Anfrage aus dem Cache beantwortet werden kann, sowie das Nachladen von Daten aus dem Backend wird durch Constraints gesteuert.

Wir verwenden allerdings nur eine bestimmte Klasse von Anfragen, nämlich PSJ-Anfragen (Projektion, Selektion, Join). In diesen Anfragen werden Gleichheitsprädikate verwendet, die auf den Satzmengen einzelner Tabellen definiert sind. Diese Gleichheitsprädikate existieren in zwei Ausprägungen:

- Eine Variante fixiert eine Spalte auf einen bestimmten Wert, beschreibt also die Eigenschaft der Satzmenge, dass sie in einer bestimmten Spalte einen bestimmten Wert enthält.
- Die andere Variante vergleicht den Wert einer Spalte einer Tabelle mit dem Wert in einer Spalte in einer anderen Tabelle, dient also zur Definition von Equi-Joins.

Im Folgenden nennen wir der Einfachheit halber nur noch die erste Variante **Gleichheitsprädikate** und die zweite Variante **Equi-Join-Prädikate**.

Im Kern haben die Prädikate der von uns unterstützen Anfragen die Gestalt einer Kette [Büh04]

$$T_1.c = \varphi_{ck} \wedge T_1.e = T_2.k \wedge \ldots \wedge T_{n-1}.e = T_n.k \qquad (1.4.1)$$

für bestimmte Folgen von Tabellen[3] T_1 bis T_n. Dabei existieren in den Tabellen bestimmte Spalten $T_i.e$ und $T_{i+1}.k$ ($1 \leq i < n$), die durch jeweils einen Equi-Join ($T_i.e = T_{i+1}.k$) verbunden sind. Außerdem wird die Spalte $T_1.c$ auf den Wert φ_{ck} fixiert. Wir können solche Prädikate in einzelne, durch Konjunktion verbundene Teilprädikate zerlegen. Einer dieser Terme ist ein Gleichheitsprädikat, der Rest sind Equi-Join-Prädikate.

Sofern wir es nicht explizit anders sagen, gehen wir im Folgenden immer davon aus, dass betrachtete Anfragen immer Prädikate mit der oben dargestellten Gestalt haben. Die Prädikate können zusätzlich mit weiteren Termen durch Konjunktion verbunden sein, um die Extension einzuschränken. Von dieser Verfeinerung werden wir im Folgenden aber abstrahieren. Außerdem werden wir der Einfachheit halber nicht mehr zwischen den Begriffen Prädikat und Anfrage unterscheiden. Mit einer Anfrage ist immer ein bestimmtes Prädikat verbunden.

Damit wir Anfragen mit solchen Prädikaten aus dem Cache beantworten können, müssen wir ihre Extensionen vollständig im Frontend speichern. Die Eigenschaft des Caches, dass die Extensionen der instanziierten Prädikate komplett gespeichert sind, wird **Prädikatvollständigkeit** genannt.

Wir wollen uns im Folgenden ansehen, wie die Extensionen dieser Prädikate aussehen, d. h. welche Sätze in den verschiedenen Tabellen T_1 bis T_n im Cache gespeichert werden müssen, um den Cache für Anfragen mit solchen Prädikaten prädikatvollständig zu machen. Dazu hangeln wir uns von links nach rechts entlang der in Formel 1.4.1 dargestellten Kette (d. h. wir betrachten die einzelnen Terme iterativ und beginnen beim Gleichheitsprädikat). Es handelt sich dabei um eine Konjunktion eines Gleichheitsprädikats mit mehreren Equi-Join-Prädikaten. Deshalb müssen in der Tabelle T_i ($i > 1$) (im Cache) alle Sätze vorhanden sein, die als Verbundpartner für die in T_{i-1} gespeicherten Sätze dienen, wobei letztere zur Beantwortung der Anfrage notwendig sind.

Wir beginnen mit dem ersten Term. Mit diesem Gleichheitsprädikat wird die Spalte $T_1.c$ auf den Wert φ_c fixiert. Es müssen also in dieser Tabelle alle Sätze im Cache sein, die den Wert φ in $T_1.c$ haben.

[3] Für $i \neq j$ müssen die Tabellen T_i und T_j nicht unterschiedlich sein. Jedoch gilt immer $T_i.e \neq T_{i+1}.k$, denn ansonsten wäre der Term „$T_i.e = T_{i+1}.k$“ eine Tautologie und wir könnten davon abstrahierten. Die Spalten $T_i.k$ und $T_i.e$ innerhalb der gleichen Tabelle dürfen ebenfalls durchaus identisch sein, gleiches gilt für $T_1.c$ und $T_1.e$.

Definitionen:

- Wir bezeichnen mit **Werteumgebung** von φ in c die Satzmenge $S_{c,\varphi}$, deren Elemente einen Wert φ in Spalte c haben.[4]
- Wenn die gesamte Werteumgebung eines Werts φ in einer Spalte c im Cache und diese nicht leer ist, nennen wir φ (in c) **umgebungsvollständig**.
- Ist die Werteumgebung von φ in c leer oder φ umgebungsvollständig, so wird φ in c **einstiegsfähig** genannt.
- Wir nennen die Menge der Werte, die im Backend in einer bestimmten Spalte gespeichert sind **dynamische Backend-Domain** (**DBD**) dieser Spalte. Im Gegensatz dazu nennen wir die Menge der dort theoretischen speicherbaren Werte **statische Backend-Domain**.
- Eine Spalte wird **komplett einstiegsfähig** genannt, wenn alle Werte aus der statischen Backend-Domain entweder einstiegsfähig oder nicht im Frontend geladen sind

In [Büh04] und anderen Arbeiten über Cache-Groups wird ein Wert „**wertvollständig**“ in einer Spalte genannt, wenn alle Sätze aus dem Backend, die diesen Wert in der Spalte haben, in der Cache-Tabelle geladen sind. Aus der dort eingeführten Definition ergibt sich allerdings auch, dass ein Wert in diesem Sinn wertvollständig ist, wenn kein Satz im Backend vorhanden ist, der diesen Wert in der betrachteten Spalte hat. Die Eigenschaft eines Wertes wertvollständig zu sein, entspricht also in unseren Begriffen der Eigenschaft einstiegsfähig zu sein.

Die Eigenschaft eines Werts φ, einstiegsfähig in einer Spalte c zu sein, beschreibt bildhaft, wie ein φ verwendet werden kann: zum Beantworten von Benutzeranfragen, bei denen ein Gleichheitsprädikat „c = φ“ die Spalte c auf den Wert φ fixiert. Bei der Sondierung des Caches (siehe Abschnitt 1.6) sprechen wir auch von „Einstieg“ in die Cache-Group.

Der Begriff „umgebungsvollständig“ kommt daher, dass die komplette Werteumgebung eines in einer Spalte c Werts φ im Cache geladen ist.

Ein umgebungsvollständiger Wert ist also immer einstiegsfähig. Diese Bedingung ist allerdings lediglich hinreichend, nicht jedoch notwendig. Ein Wert φ kann einstiegsfähig aber nicht umgebungsvollständig sein, wenn seine Werteumgebung leer ist. Dieser Wert liegt dann nicht in der DBD der betrachtenden Spalte.

Wenden wir uns wieder der Extension des in Formel 1.4.1 dargestellten Prädikats zu. Damit der Cache für die betrachtete Anfrage prädikatvollständig sein kann, muss also die Werteumgebung von φ_{ck} in T_1.c sein. Diese Werteumgebung entspricht der Extension des Gleichheitsprädikats (d. h. des ersten Terms in der Kette). Der Wert φ_{ck} muss einstiegsfähig, aber nicht unbedingt umgebungsvollständig sein: Wenn die Werteumgebung von φ_{ck} in T_1.ck leer ist, ist somit die gesamte Extension des Prädikats leer und der Cache für das Prädikat prädikatvollständig.

[4] Für eine Erklärung der Schreibweise und der verwendeten Symbole für Spalten, Werte und Satzmengen sei auf Anhang B „Schreibweisen und Symbolverzeichnis“ verwiesen.

Betrachten wir nun die Satzmenge, die in der Tabelle T_2 im Cache sein muss, damit wir das in Formel 1.4.1 dargestellte Prädikat aus dem Cache beantworten können. Gefordert ist, dass alle Sätze, die dass Prädikat „$T_1.e = T_2.k$“ erfüllen, im Cache sind. Das sind alle Sätze, die einen Wert in der Spalte $T_2.k$ haben, der im Cache in der Spalte $T_1.e$ vorkommt. Mit anderen Worten: Alle Werte φ_e, die in $T_1.e$ im Cache geladen sind, müssen in $T_2.k$ einstiegsfähig sein. Werte unter diesen, deren Werteumgebung in $T_2.k$ nicht leer ist, müssen also insbesondere umgebungsvollständig sein.

Diese Bedingung können wir auf die weiteren Tabellen T_i fortsetzen. Jeder Wert, der in $T_i.e$ im Cache vorkommt, muss in $T_{i+1}.k$ einstiegsfähig sein.

1.5 Cache-Groups für Gleichheitsprädikate

Um zu garantieren, dass Anfragen mit Prädikaten der Gestalt, wie sie in Formel 1.4.1 dargestellt ist, aus dem Cache beantwortet werden können, setzt der Cache-Manager bei dem in dieser Arbeit vorgestellten Caching-Verfahren Constraints durch. Welche Arten von Anfragen mit der dargestellten Form aus dem Cache beantwortet werden können, wird dabei durch Cache-Groups beschrieben. Die Cache-Groups bilden also die statische Struktur (d. h. die im Cache vorhandenen Tabellen und die geltenden Constraints), die die unterstützten Prädikatstypen definiert.

Eine **Cache-Group** besteht aus einer Menge von **Cache-Tabellen** sowie einer Menge von **Constraints**. Die Cache-Tabellen repräsentieren eine Auswahl von Tabellen aus der Backend-DB im Cache und stimmen mit ihnen im Schema überein.[5] Ein Cache-Tabelle enthält stets eine Teilmenge vollständiger Sätze aus der ihr zugeordneten Tabelle im Backend [Büh04].

Die **Constraints** einer Cache-Group beschreiben die Inhalte sowie die Abhängigkeiten zwischen den Inhalten der einzelnen Cache-Tabellen und definieren so gültige Zustände des Caches. Constraints können vom Datenbank-Administrator mit dem Ziel definiert werden, festzulegen, welche Anfragetypen vom Cache unterstützt werden. Je nach dem, welche Anfragetypen vom Cache unterstützt werden sollen, können in den Cache-Groups unterschiedliche Constraints definiert sein.

In einem gültigen Zustand kann vom Cache-Manager entschieden werden, ob eine Anfrage aus dem Cache beantwortet werden kann. Indem im Cache die definierten Constraints durchgesetzt werden, wird garantiert, dass die Extensionen vollständig im Cache sind und der Cache somit immer prädikatvollständig ist.

In dieser Arbeit betrachten wir lediglich Cache-Groups für Gleichheitsprädikate (wobei wir hier wieder Equi-Join-Prädikate einschließen), die die in Formel 1.4.1 gezeigte Gestalt haben. Wir sprechen im Folgenden deswegen auch allgemein nur noch von Cache-Groups, wenn wir Cache-Groups für Gleichheitsprädikate meinen.

[5] Allerdings bezeichnen wir mit dem Begriff Cache-Tabelle in späteren Abschnitten dieser Arbeit wechselweise sowohl die eigentliche Cache-Tabelle (d. h. die Tabelle in der Frontend-DB) als auch die Menge ihrer Spalten. Das gemeinte ergibt sich dabei jeweils aus dem Kontext.

Konkret existieren in einer Cache-Group zwei Arten von Constraints: Cache-Key-Constraints und **referentielle Cache-Constraints** (RCCs).

Cache-Key-Constraints werden für Spalten in Cache-Tabellen definiert. Spalten, für die eine Cache-Key-Constraint definiert ist, heißen Cache-Keys und erfüllen folgende Eigenschaften:

- Alle im Cache-Key vorhandenen Werte müssen einstiegsfähig sein. Diese Eigenschaft nennen wir **Einstiegspunkteigenschaft**. Jede Spalte mit Einstiegspunkteigenschaft ist also immer komplett einstiegsfähig, d. h. jeder Wert aus der DBD ist einstiegsfähig. Insbesondere muss die Werteumgebung eines Werts, der in einem Cache-Key vorkommt, vollständig in der Cache-Tabelle des Cache-Keys geladen sein.
- Die zweite so genannte **Füllpunkteigenschaft** besagt, dass Anfragen, die den Cache-Key auf einen Wert φ fixieren, bei einem **Cache-Miss** (d. h. wenn die Anfrage nicht aus dem Cache beantwortet werden kann) dazu führen, dass die Werteumgebung von φ im Cache-Key in den Cache geladen wird. Man beachte, dass durch diese Eigenschaft allein nicht garantiert würde, dass die Anfrage zukünftig aus dem Cache beantwortet werden kann. Dazu müssen noch weitere Sätze in anderen Cache-Tabellen geladen werden, damit Verbundpartner bei Equi-Joins gefunden werden. Für dieses Verhalten wiederum sind die RCCs zuständig. Ein Cache-Key stellt somit einen **Füllpunkt** einer Cache-Group dar.

Aufgrund der Einstiegspunkteigenschaft können wir prüfen, ob bestimmte Benutzeranfragen mit einem Gleichheitsprädikat, das den Cache-Key auf einen Wert fixiert, aus dem Cache beantwortet werden können. Zumindest die Extension dieses Terms ist garantiert im Cache. Die Einstiegspunkteigenschaft ist durch die Definition eines Cache-Keys für die Spalte explizit gegeben. Spalten können aber auch implizit die Einstiegspunkteigenschaft besitzen, so z. B. Unique-Spalten (d. h. Spalten, bei denen in jedem Satz ein anderer Wert vorkommt).

Durch die Füllpunkteigenschaft erhält der Cache sein adaptives Verhalten. Ein Cache-Miss führt immer dazu, dass Anfragen, welche aufgrund seiner Struktur (d. h. den verwendeten Equi-Join-Prädikaten) prinzipiell unterstützt werden, zukünftig aus dem Cache beantwortet werden können; zumindest solange ihre Extension nicht wieder aus dem Cache verdrängt wird.

Prinzipiell erlauben wir für eine gegebene Cache-Group lediglich die Existenz von Cache-Keys in einer ausgezeichneten Cache-Tabelle, der so genannten Wurzel-Tabelle. Durch das Verschmelzen von mehreren Cache-Groups zu einer **Cache-Föderation** können jedoch Cache-Keys in verschiedenen Cache-Tabellen der Föderation existieren. Cache-Föderationen werden gebildet, um ein redundantes Speichern gleicher Daten von verschiedenen Cache-Groups zu vermeiden. In Cache-Föderationen teilen sich also mehrere Cache-Groups die gleichen Cache-Tabellen (und Constraints).

RCCs definieren Bedingungen, die zwischen Spalten in der Cache-Group herrschen müssen. Durch diese Constraints wird festgelegt, welche Prädikatstypen vom Cache unterstützt werden. Sie schränken also das Schema der verwendeten parametrisierten Prädikate ein. Wie wir gesehen haben, haben die unterstützten Prädikate immer die Form einer Kette, wie sie in Formel 1.4.1 dargestellt ist. Das Gleichheitsprädikat fixiert einen Cache-Key auf einen Wert. Die Equi-Join-Prädikate stellen Equi-Joins zwischen verschiedenen Tabellen her. Die RCCs

sorgen dafür, dass diese Equi-Joins vom Cache unterstützt werden. Dafür werden durch die Constraints zwei Spalten über eine Werte-Beziehung verknüpft.

Definition: Eine RCC e→k von einer Spalte e zu einer Spalte k ist genau dann erfüllt, wenn alle Werte, die im Cache in e vorkommen, in k einstiegsfähig sind.[6]

Wenn die Extensionen der Terme „$T_1.c = \varphi$“ bis „$T_{i-1}.e = T_i.k$“ – also bis vor einem betrachtenden Equi-Join-Prädikat – im Cache sind, so wird durch eine RCC $T_i.e \rightarrow T_{i+1}.k$ sichergestellt, dass auch die Extensionen aus T_{i+1}, die zur Beantwortung der Anfrage benötigt werden, im Cache sind.

Um unsere betrachtenden Anfragen zu unterstützen, muss also für jedes Equi-Join-Prädikat „$T_i.e = T_{i+1}.k$“ eine entsprechende RCC $T_i.e \rightarrow T_{i+1}.k$ im Cache gelten.

Man beachte, dass ebenfalls allein durch die RCCs eine korrekte Beantwortung von Equi-Join-Anfragen „$T_i.e = T_{i+1}.k$“ (ohne weitere Einschränkung durch Konjunktion mit anderen Prädikaten) aus dem Cache nicht gewährleistet werden kann. Einige der Sätze im Backend von T_i, die einen Verbundpartner in T_{i+1} im Cache haben, sind eventuell nicht in der Cache-Tabelle T_i geladen. Dadurch allerdings, dass unsere Anfragen immer auch ein Gleichheitsprädikat „$T_1.c = \varphi$“ beinhalten, können wir die Cache-Tabelle T_1 als „Anker“ benutzen und die Verbundpartner für den Equi-Join „$T_1.e = T_2.k$“ aus der Cache-Tabelle T_2 holen, wenn eine RCC „$T_1.e \rightarrow T_2.k$“ gilt. Dadurch ist wiederum T_2 „verankert“. Somit können gegebenenfalls weitere Verbundpartner für ein Equi-Join-Prädikat „$T_2.e = T_3.k$“ aus dem Cache zur Beantwortung herangezogen werden usw.

Durch die RCC-Eigenschaft kann die Einstiegspunkteigenschaft der Cache-Keys mit Hilfe von **Selbst-RCCs** beschrieben werden. Diese besonderen RCCs verbinden eine Spalte $T_i.c$ mit sich selbst, die RCC hat also die Gestalt $T_i.c \rightarrow T_i.c$. Bei einer Spalte mit einer solchen Selbst-RCC gilt ebenfalls die Einstiegspunkteigenschaft. Wir werden im Folgenden also die Einstiegspunkteigenschaft der Cache-Keys durch Selbst-RCCs modellieren.

In der Literatur über Cache-Groups (unter anderem in [HB04a]) werden verschiedene Arten von RCCs unterschieden, je nach dem welche Typen von Spalten durch eine RCC verbunden werden:

- Eine RCC, die bei einer Unique-Spalte anfängt, wird „**Member-Constraint**“ (MC) genannt,
- eine RCC, die bei einer Nicht-Unique-Spalte anfängt, aber bei einer Unique-Spalte endet, wird „**Owner-Constraint**“ (OC) genannt und
- eine RCC, die zwei Nicht-Unique-Spalten miteinander verbindet, wird „**Cross-Constraint**“ (XC) genannt.

Neben explizit definierten RCCs können auch weitere implizit geltende RCCs im Cache existieren. Diese RCCs werden **Optimierungs-RCCs** genannt. Diese zusätzlich vorhandenen RCCs ergeben sich als Folge der explizit definierten RCCs und ermöglichen die Unterstüt-

[6] Wir werden später Spalten am Anfang einer RCC Konsumenten und am Ende einer RCC Erzeuger nennen (siehe Abschnitt 2.2), daher die Symbole e bzw. k.

zung weiterer Equi-Join-Prädikate. Als Beispiel für eine Optimierungs-RCCs sei eine OC $T_1.k \rightarrow T_2.e$ angeführt, für die eine korrespondierende, explizit definierte MC $e \rightarrow k$ existiert, wenn an T_2 keine weitere RCC endet und nur zur Einhaltung der OC benötigten Sätze vom Cache-Manager in T_2 geladen werden. Dies gilt, denn jeder in $T_1.k$ geladene Wert φ ist umgebungsvollständig und damit einstiegsfähig sowohl in $T_1.k$ als auch in $T_2.e$: Die Spalte $T_1.e$ ist unique, wodurch jeder geladene Wert (also insbesondere auch φ) automatisch umgebungsvollständig ist, und φ ist nur deshalb in $T_2.k$ geladen, weil der Cache-Manager seine Werteumgebung in T_2 geholt hat, um ihn einstiegsfähig zu machen und somit die OC durchzusetzen (ein anderer Weg, über den φ in $T_2.k$ eingeführt werden könnte, ist nach Voraussetzung ausgeschlossen). Somit ist φ in $T_2.k$ umgebungsvollständig, wenn der Wert in dieser Spalte geladen ist.

Verschiedene Arbeiten über Cache-Group setzen sich mit dem Thema Optimierungs-RCCs auseinander z. B. [HB04a]. Da für diese Arbeit Optimierungs-RCCs allerdings nicht interessant sind, weil sie das Füllverhalten einer Cache-Group nicht beeinflussen (der Cache verhält sich allein auf Grund der deklarierten RCCs bezüglich des Füllverhaltens so, als wären die Optimierungs-RCCs definiert), wird von diesen abstrahiert. Wir sprechen deswegen in Zukunft nur noch von definierten RCCs (inklusive der Einstiegspunkteigenschaft in definierten Cache-Keys in Form von Selbst-RCCs) als RCCs.

1.6 Abgrenzung zu anderen Arten des DB-Caching

Nachdem wir uns bisher mit den in dieser Arbeit betrachteten Objekten auseinander gesetzt haben, nämlich Cache-Groups für Gleichheitsprädikate, wollen wir in diesem Abschnitt kurz betrachten, welche Alternativen prinzipiell für das Datenbank-Caching zur Verfügung stehen. Zunächst wollen wir uns aber Beispiele für Systeme ansehen, die Cache-Groups für Gleichheitsprädikate in der Praxis verwenden.

Eines dieser Systeme, welche Cache-Groups verwendeten stellt das System DBCache [IBM04] der Firma IBM dar. DBCache ist ein Prototyp eines Systems, dass DB-Caching-Fähigkeiten in ein vollwertiges DBMS aufnimmt. Als DBMS wird hierbei das System DB2 von IBM eingesetzt. Bei DBCache wird eine übliche DB2-Instanz in einen transparenten DB-Cache-Manager verwandelt. Hierfür wurde der zugrunde liegende Engine-Code angepasst und bereits vorhandene Funktionalität für föderierte Datenbanken wirksam umgesetzt. [ABK+03]

Ein anderes System, welches Cache-Groups in der Praxis einsetzt, ist das Produkt TimesTen/Cache [TT04] der Firma TimesTen. Der vom TimesTen ursprünglich verwendete Begriff Cache-Group beschreibt eine Auswahl von Tabellen im Frontend (also in unserem Sinn Cache-Tabellen), die in einer baumartigen Struktur über Member-Constraints (d. h. RCCs, die bei einer Unique-Spalte beginnen) verbunden sind. Die gespeicherten Bäume sind allerdings nicht überlappend und werden eindeutig durch einen Schlüssel in der Wurzel-Tabelle identifiziert (ein Wert in einer Unique-Spalte, z. B. dem Primärschlüssel; in unserem sind der Cache-Key). Zu einem Datensatz in der Wurzeltabelle sind also die zugehörigen Sätze in den anderen Tabellen im Cache zuzuordnen. Wir werden diese Klasse von Cache-Groups später als „baumartig“ bezeichnen.

Wie wir gesehen haben, können für das Datenbank-Caching auch andere Alternativen eingesetzt werden.

Hier zu nennen wäre noch einmal das bereits erwähnte Volltabellen-Caching, bei dem alle Sätze ausgewählter Tabellen im Frontend vorgehalten werden. Wegen der hohen Replikations- und Wartungskosten bei häufig aktualisierten Tabellen ist vom Einsatz dieser Technologie allgemein abzuraten.

Eine andere Alternative wäre der Einsatz von materialisierten Sichten. Diese können entweder explizit vom Datenbank-Administrator deklariert werden oder adaptiv an sich änderte Anfragecharakteristika gebildet werden. In [LGZ04] ist ein solches System beschrieben,.

Prinzipiell kommt es bei materialisierten Sichten vor, dass Daten im Cache mehrfach gehalten werden, da sie in verschiedenen Sichten verwendet werden. Eine Alternative hierfür stellen replikationsfreie Sichten dar. Indem man den Zusammenhang zwischen verschiedenen Sichten, etwa deren Bezug auf gemeinsame Tabellen, ausnutzt, lassen sich Sichten in gemeinsamen Strukturen im Cache materialisierten. Ein konkretes Verfahren hierfür, zumindest in der Zielvorstellung, stellt das System DBProxy [APT+03] der Firma IBM dar. DBProxy soll adaptiv auf sich ändernde Anfragecharakteristika reagieren und dabei eine große Zahl von überlappenden und sich dynamisch ändernden materialisierter Sichten speichern. Neue Sichten werden auf Anfrage automatisch hinzugefügt und eventuell wieder verworfen (und die Daten verdrängt), wenn sie eine Zeit lang nicht benötigt werden, um Speicherplatz zu sparen.

Eine weitere Alternative zu Cache-Groups mit Gleichheitsprädikaten ist die Verwendung von Cache-Groups für andere parametrisierte Prädikate. Eine Möglichkeit stellt das Caching der Extensionen von Bereichsprädikaten statt Gleichheitsprädikaten dar. Ein solches Prädikat prüft einen Wert daraufhin, ob er zwischen einer gegebenen oberen und unteren Schranke liegt. In einem Einstiegspunkt (d. h. eine Spalte auf der ein Bereichsprädikat definiert ist, das vom Cache unterstützt werden soll) muss jeder Wert zwischen (und inklusive) der oberen und unteren Schranke einstiegsfähig sein. Nur so kann vom Cache-Manager entschieden werden, ob die Anfrage aus dem Cache beantwortet werden kann. Um die Sondierung zu realisieren, können vom Cache-Manager geordnete Listen der einstiegsfähigen Bereiche geführt werden, wobei angrenzende Bereiche automatisch zu einem einzigen, größeren Bereich vereinigt werden.

Vollkommen andersartig ist die Möglichkeit, die Daten, welche in einem Datenbank-Puffer gehalten werden lokal vorzuhalten. Hierbei wird kein vollständiges DBMS zur Verwaltung des Caches eingesetzt, sondern lediglich die Schichten oberhalb und inklusive des DB-Puffers. Werden benötigte Daten nicht in Kacheln des DB-Puffers im Frontend gefunden, so werden diese aus dem Backend angefordert.

Solche Lösungen existieren ebenfalls bereits im Kontext von verteilten DBMS. Dabei muss auf Besonderheiten wie die Eigentümer der Seiten und Kohärenz des Caches geachtet werden. Schliesslich können mehrere DB-Puffer verteilt nebeneinander die gleichen Seiten zwischengespeichert haben. Außerdem werden so eventuell viele, eigentlich nicht benötigten Sätze, im Cache vorgehalten, da wir nicht die notwendige Semantik auf den Ebenen unter dem DB-Puffer zur Verfügung haben. Des Weiteren kann die Menge der zu zwischenzuspeichernden Objekte bei diesem Verfahren nicht eingeschränkt werden.

1.7 Anfrageverarbeitung und Füllverhalten

Nachdem wir das Constraint-basierte Datenbank-Caching mit Hilfe von Cache-Groups für Gleichheitsprädikate formal definiert und die Eigenschaften der Constraints untersucht haben, wollen wir nun betrachten, wie Anfragen aus dem Cache beantwortet werden. Außerdem umreißen wir kurz das Füllverhalten des Caches nach einem Cache-Miss, wenn also eine ankommende Anfrage nicht aus dem Cache beantwortet werden kann. Dieses Wissen soll dazu dienen, dass wir die Problematik verstehen, welche in dieser Arbeit hauptsächlich thematisiert wird: die Modellierung des Füllverhaltens von Cache-Groups.

In einem ersten Schritt muss natürlich vom Cache-Manager entschieden werden, ob die Anfrage aus dem Cache beantwortet werden kann. Diesen Schritt nennen wir **Sondierung** des Caches. Kann die Anfrage aus dem Cache beantwortet werden (d. h. bei einem **Cache-Hit**), so wird die Antwortmenge direkt aus dem Cache an den Benutzer (bzw. die benutzende Anwendung) geschickt. Der Zustand des Caches wird bei einem Cache-Hit nicht verändert.

Führt die Sondierung zu einem negativen Ergebnis, so wird die Anfrage an den Backend-Server weitergeleitet. Gleichzeitig wird gegebenenfalls durch das Laden zusätzlicher Sätze in die Cache-Tabellen dafür gesorgt, dass eine gleiche Anfrage zukünftig aus dem Cache beantwortet werden kann. Dieser Vorgang des Nachladens neuer Sätze in den Cache nach einem Cache-Miss nennen wir **Füllzyklus**.

Betrachten wir kurz die Sondierung des Caches. Wir wollen die Sondierung und Anfragebeantwortung an dieser Stelle sehr vereinfacht darstellen, da wir uns hauptsächlich mit dem Füllverhalten des Caches beschäftigen wollen.

Bei der Sondierung wird zunächst geprüft, ob die gegebene Anfrage überhaupt vom Typ her vom Cache unterstützt wird. Dazu ist es notwendig, dass für jedes Equi-Join-Prädikat „$T_i.e = T_{i+1}.k$“ in der Anfrage eine RCC $T_i.e \rightarrow T_{i+1}.k$ im Cache gilt. Natürlich können prinzipiell nur Anfragen der Gestalt aus Formel 1.4.1 beantwortet werden. Werden diese Anforderungen an den Prädikatstyp von der gegebene Anfrage erfüllt, so wird geprüft, ob sich die Extension des Prädikats im Cache befindet. Hierfür wird lediglich getestet, ob er im Gleichheitsprädikat der Anfrage ($T_1.c = \varphi$) referenzierte Wert φ in der Spalte $T_1.c$ im Cache ist und ob $T_1.c$ komplett einstiegsfähig ist. Bei Spalten, die nicht ohnehin implizit komplett einstiegsfähig sind, muss es sich um einen explizit definierten Cache-Key handeln, damit vom Cache-Manager dadurch, dass er die Einstiegspunkteigenschaft durchsetzt, garantiert wird, dass die Spalte komplett einstiegsfähig ist.

Bei einem positiven Ausgang der Sondierung wird die Anfrage aus dem Cache beantwortet. Alle zur Beantwortung benötigten Sätze stehen in den verschiedenen Cache-Tabellen zur Verfügung.

Bei [ABK+03] funktioniert die Sondierung etwas flexibler als hier dargestellt: Es können auch mehrere Einstiegspunkte verwendet werden und auch nur Anfangsteile von „Anfrageketten“ können durch RCCs unterstützt werden. Nichts desto trotz liegt im Kern nur die einfache Kette nach Formel 1.4.1 zugrunde. In dem dort vorgestellten Verfahren zur Anfrageoptimierung wird ein Plan generiert, der auch Referenzen auf Backend-Tabellen enthalten kann und verteilt ausgewertet wird. In [Büh04] finden wir darüber hinaus eine noch flexiblere Art der

Sondierung, da bei dem dort vorgestellten Verfahren mehr Spalten als Einstiegspunkte zur Verfügung stehen.

Bei einem Cache-Miss, d. h. einem negativem Ergebnis der Sondierung, wird die Anfrage an den Backend-Server weitergeleitet und von dort aus beantwortet. Der Benutzer bzw. die benutzende Anwendung bekommt allerdings von dieser Umleitung nichts mit, da die Antwort über den Cache an den Benutzer geschickt wird.

Wir können in einem Cache-Miss-Fall zwei Situationen unterscheiden:

- Handelt es sich bei der im Gleichheitsprädikat referenzierten Spalte um einen Füllpunkt, so wird ein neuer Füllzyklus angestoßen. Die Werteumgebung des referenzierten Werts wird in die Cache-Tabelle des Füllpunkts geladen und dadurch eventuell verloren gegangene RCC-Bedingungen durch den Cache-Manager wieder hergestellt. Dadurch wird der Cache stets in einem gültigen Zustand gehalten. Das Nachladen der Werteumgebung des referenzierten Werts und die Wiederherstellung der RCC-Bedingungen kann als atomare Transaktion im Sinne des ACID-Paradigmas gesehen werden.
- Wenn die im Gleichheitsprädikat referenzierte Spalte kein Füllpunkt ist, so bleibt der Zustand des Caches unverändert. Die gleiche Anfrage könnte damit zukünftig auch nicht aus dem Cache beantwortet werden, sondern müsste wiederholt an den Backend-Server weiterleitet werden.

Wenn die Extension der ankommenden Anfrage nicht im Cache ist und die im Gleichheitsprädikat referenzierte Spalte ein Füllpunkt ist, wird ein neuer Füllzyklus für den Wert (im Folgenden Cache-Key-Wert) im Füllpunkt (bzw. Cache-Key) angestoßen. Wir wollen im Folgenden kurz betrachten, wie ein solcher Füllzyklus abläuft.

Prinzipiell beginnt ein Füllzyklus damit, dass die Werteumgebung des Cache-Key-Werts in die Cache-Tabelle des Füllpunkts geladen wird. Hierdurch wird die Füllpunkteigenschaft des Cache-Keys vom Cache-Manager durchgesetzt. Da die gesamte Werteumgebung geladen wird, ist der Cache-Key nach diesem Vorgang immer noch komplett einstiegsfähig, wenn er vorher komplett einstiegsfähig war. Die Einstiegspunkteigenschaft bleibt also auch erhalten.

Wir nennen den Vorgang, bei dem neue Sätze in eine Cache-Tabelle geladen werden, **Füllschritt**. Durch einen solchen Füllschritt werden allgemein neue Werte in die verschiedenen Spalten der Cache-Tabelle eingeführt. Wir unterscheiden bei der Ursache der Einführung von Werten in eine Spalte bei einem Füllschritt zwei Möglichkeiten:

- Werden Sätze in die Cache-Tabelle geladen, um eine Bedingung/Eigenschaft für eine Spalte zu erfüllen (z. B. die Füllpunkteigenschaft des Cache-Keys) und dadurch Werte in eine Spalte eingeführt, so sagen wir, die Werte werden in die betrachtete Spalte **induziert**.
- Das Einführen von Werten in andere Spalten der Cache-Tabelle durch das Laden der Sätze wird **Schmuggeln** genannt. Zwischen den Spalten, in denen Werte induziert werden, und den anderen Spalten herrscht in jeder Cache-Tabelle eine so genannte **Schmuggler-Beziehung**.

Man beachte, dass ein Wert, der bei einem Füllschritt induziert wird, vorher bereits in der Spalte im Cache vorhanden gewesen sein kann. Der Wert kann zunächst in die betrachtete Spalte geschmuggelt worden sein. Da er aber nicht umgebungsvollständig und damit einstiegsfähig war, mussten weitere Sätze aus dem Backend in die Cache-Tabelle geladen werden. Somit wurde der Wert zunächst geschmuggelt und danach induziert.

Durch das Induzieren des Cache-Key-Werts in den Füllpunkt werden Sätze in seine Cache-Tabelle geladen, die wiederum zu einem Schmuggeln von Werten in die anderen Spalten führen.

Das Einführen von Werten in Spalten hat prinzipiell zur Folge, dass gegebenenfalls RCCs, die an den Spalten abgehen, nicht mehr gelten. Denn durch das Einführen der Werte sind diese in den Spalten am Anfang der RCC geladen, in den Spalten an ihrem Ende aber möglicherweise nicht einstiegsfähig.

Wir nennen eine Spalte k am Ende einer RCC e→k **ungesättigt**, wenn mindestens ein Wert dort nicht einstiegsfähig ist, der in e geladen ist.

Wir können uns diese Eigenschaft bildhaft vorstellen, indem wir uns vor Augen halten, dass zwischen den in der Spalte am Anfang der RCC geladenen und in der Spalte am Ende der RCC umgebungsvollständigen Werte ein Ungleichgewicht herrscht, dass ausgeglichen werden will.

Man beachte, dass eine Spalte c mit einer Selbst-RCC (also insbesondere auch Cache-Keys wegen ihrer Einstiegspunkteigenschaft) auch dann ungesättigt sind, wenn ein Wert in ihr geladen, aber nicht umgebungsvollständig ist. Hier handelt es sich also um einen Spezialfall.

Sind nach dem ersten Füllschritt (zum Laden der Werteumgebung des Füllpunkts in seine Cache-Tabelle) Spalten in irgendeiner Cache-Tabelle ungesättigt, so muss für diese jeweils mindestens ein weiterer Füllschritt ausgeführt werden. Bei diesem Füllschritt werden dann Werte induziert (und damit Sätze in die Cache-Tabelle geladen), um die RCC-Eigenschaft durchzusetzen[7]. Hierdurch werden wiederum Werte in andere Spalten geschmuggelt und weitere Spalten ungesättigt.

In einem Füllzyklus werden solange Füllschritte ausgeführt, bis keine Spalte in der Cache-Group mehr ungesättigt ist. Anschließend befindet sich die Cache-Group wieder in einem gültigen Zustand, so dass wir feststellen können, ob weitere Anfragen aus dem Cache beantwortet werden können. Wir sagen auch, dass eine Cache-Group in einem gültigen Zustand – d. h. nach einem einzelnen betrachteten Füllzyklus – **stabil** ist. Dieser Zustand wird garantiert nach einer endlichen Anzahl von Füllschritten erreicht; ein Füllzyklus terminiert also immer. Spätestens wenn alle Sätze aus den Backend-Tabellen in den korrespondierenden Cache-Tabellen sind, ist keine Spalte mehr ungesättigt und die Cache-Group somit stabil. Während eines Füllzyklus kommen die verschiedenen Cache-Tabellen nach und nach in einen Zustand,

[7] Ein Füllschritt muss aber nicht unbedingt dazu führen, dass automatisch die vorher ungesättigte Spalte anschließend gesättigt ist. Wir können im Extremfall auch nur einen der Werte induzieren. In Abschnitt 2.3 werden wir uns näher mit verschiedenen Arten von Füllschritten und deren Verwendung zur Modellierung des Füllverhaltens von Cache-Groups beschäftigen.

bei dem sie keine weiteren Füllschritte mehr ausführen. Wir nennen eine Cache-Tabelle **vollständig bevölkert**, wenn von ihr keine weiteren Füllschritte mehr ausgeführt werden.[8] Wir stellen uns vor, dass in eine vollständig bevölkerte Cache-Tabelle keine weiteren Sätze mehr geladen werden. Dementsprechend ist die Population abgeschlossen. Wenn alle Cache-Tabellen vollständig bevölkert sind, ist die Cache-Group also stabil und der Füllzyklus beendet.

Exzessives Füllverhalten

In gewissen Konstellationen führen die Constraints zu einer negativem Nebenwirkung. Dies ist z. B. dann der Fall, wenn vom Cache-Manager die Einstiegspunkteigenschaft für zwei Nicht-Unique-Cache-Keys T.a und T.b in der gleichen Cache-Tabelle T durchgesetzt werden muss.

Nehmen wir an, wir führen einen Füllschritt aus, um die Werteumgebung eines Werts φ im Cache-Key T.a in die Cache-Tabelle zu laden. Durch diesen Füllschritt werden Werte in den anderen Cache-Key T.b geschmuggelt, so dass dieser ungesättigt wird, weil seine Einstiegspunkteigenschaft nicht mehr erfüllt ist: Die geschmuggelten Werte sind im Allgemeinen nicht umgebungsvollständig, da T.b nicht unique ist. Der Cache-Manager führt also mindestens einen weiteren Füllschritt aus, um die Einstiegspunkteigenschaft für T.b durchzusetzen. Während diesen Füllschritten werden wiederum Werte in T.a geschmuggelt, die wiederum allgemein nicht umgebungsvollständig sind. Nun ist T.a also ungesättigt und der Cache-Manager muss weitere Füllschritte ausführen. Im Extremfall können durch dieses exzessive Füllverhalten alle Sätze aus dem Backend in die Cache-Tabelle T geladen werden.

Beim Design von Cache-Groups sollte also prinzipiell darauf geachtet werden, dass Constraints vermieden werden, die ein solches – **rekursives Laden** genanntes – Füllverhalten provozieren. Bei Cache-Groups, bei denen ein rekursives Laden aufgrund der definierten Constraints prinzipiell nicht vorkommen kann, wird **sicher** genannt.

Beim rekursiven Laden kann es im Extremfall dazu kommen, dass der gesamte Inhalt der Backend-Tabelle in die korrespondierende Cache-Tabelle geladen wird. De facto hätten wir es also mit einem Volltabellen-Caching zu tun, was wir auf Grund der hohen Kosten durch Instandhaltung und Aktualisierung der Tabellen vermeiden möchten. Außerdem führt ein solcher exzessiver Füllzyklus zu einer steigenden Systemlast im Frontend und im Backend, was wir auch prinzipiell vermeiden möchten.

Um ein rekursives Laden durch ein entsprechendes Design der Cache-Group im vorhinein ausschließen zu können, wollen wir kurz untersuchen, welche Voraussetzungen im Allgemeinen zu einem solchen exzessivem Füllverhalten führen. Wir können dann bei der Definition unserer Constraints diese Voraussetzungen ausschließen.

Eine Cache-Group ist immer dann sicher, wenn jede Spalte für eine dort ankommende RCC jeweils nur einmal ungesättigt werden kann und nur jeweils ein Füllschritt, der die betreffende

[8] Spätestens wenn alle Werte aus der DBD einer Spalte geladen sind, kann diese Spalte nicht mehr ungesättigt sein. Dieser Zustand wird irgendwann für alle Spalten erreicht und damit ist die Cache-Tabelle dann vollständig bevölkert.

RCC-Eigenschaft wiederherstellt, für die Cache-Tabelle ausgeführt werden muss. Wir können also bei der Wurzel-Tabelle der Cache-Group anfangen und den ersten Füllschritt ausführen, um die Werteumgebung des Cache-Key-Werts in den Cache zu holen. Anschließend führen die Nachfolger-Cache-Tabellen (das sind Cache-Tabellen, die über eine RCC erreicht werden können) eine endliche Anzahl von Füllschritten aus, anschließend deren Nachfolger usw..

Offensichtlich spielen Zyklen dabei eine wichtige Rolle, dass eine Cache-Group unsicher wird.

Wir bezeichnen als **RCC-Zyklen** zusammenhängende, geschlossene Pfade von RCCs, die Cache-Tabellen miteinander verbinden [HB04a]. Ein RCC-Zyklus heißt **homogen**, wenn in jeder Cache-Tabelle lediglich eine Spalte besucht wird, wie z. B. in Abbildung 3 links dargestellt. Dahingegen heißt ein RCC-Zyklus **heterogen**, wenn in mindestens einer besuchten Cache-Tabelle zwei Spalten am Zyklus beteiligt sind, z. B. Abbildung 3 rechts.

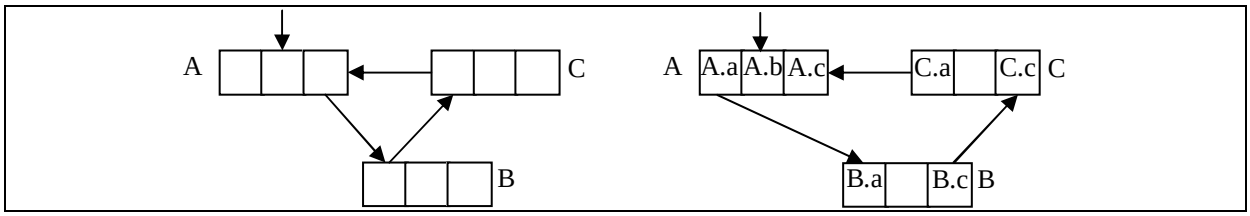

Abbildung 3: Ausschnitte aus Cache-Groups: homogener Zyklus (links) und heterogener Zyklus (rechts)

Heterogene RCC-Zyklen können allgemein (also abgesehen von Ausnahmen) zu einem exzessiven Füllverhalten führen. Wir betrachten zur Begründung einen Ausschnitt aus einem Füllzyklus in der Cache-Group aus Abbildung 3 rechts, der damit beginnt, dass ein Füllschritt für A ausgeführt wird. Außerdem gehen wir davon aus, dass die Spalten A.a und A.c nicht unique sind. Durch den ersten Füllschritt werden Werte in A.b induziert, was dazu führt, dass Werte in A.a geschmuggelt werden. Die Spalte B.a wird dadurch ungesättigt. In einem zweiten Füllschritt werden Werte in B.a induziert, um die RCC A.a→B.a durchzusetzen. Aufgrund dieses Füllschritts werden Werte in B.c geschmuggelt und C.c wird ungesättigt; im nächsten Füllschritt werden Werte in C.c induziert, um die RCC-Eigenschaft durchzusetzen. Der Füllzyklus geht dann wieder weiter bei A, da nun A.c ungesättigt ist. Schließlich ist B.a wieder ungesättigt, es wird ein Füllschritt in B ausgeführt und dadurch Werte in B.c geschmuggelt, was die Werte weiter entlang des Zyklus „laufen lässt“. Der Füllzyklus wird erst dann beendet, wenn durch einen Füllschritt keine Spalte mehr ungesättigt ist; dabei kann es sein, dass dies erst dann der Fall ist, wenn alle Sätze aus dem Backend in den Cache-Tabellen sind.

1.8 Quantitative Analyse

Wie wir gesehen haben, kann es bei einem unbedachten Design von Cache-Groups zu unvorteilhaften Nebenwirkungen durch die definierten Constraints kommen. Hauptsächlich heterogene RCC-Zyklen verursachen dabei ein exzessives Laden von Sätzen aus dem Backend, wodurch es im Extremfall zu einem Volltabellen-Caching kommen kann. Dies würde natürlich aufgrund der hohen Kosten für Wartung und Pflege der im Cache gehaltenen Daten die

Vorteile des Caching (also z. B. kürzere Antwortzeiten, höhere Ausfallsicherheit und geringere Netzwerklast) zunichte machen oder zumindest mindern.

Wünschenswert ist also eine Art „Cache-Advisor“, der einem Datenbank-Administrator zur Definition bestimmter RCCs raten kann. Auch sinnvoll wäre es, wenn die Veränderung des Füllverhaltens einer Cache-Group oder -Föderation durch das Hinzufügen einer neuen Constraint durch den Cache-Advisor überprüft werden kann. Dazu könnte der Cache-Advisor das Füllverhalten der Cache-Group vor und nach der Definition einer Constraint analysieren und so vor der Definition einer „gefährlichen“ Constraint warnen.

Ein wichtiger Schritt zur Entwicklung eines solchen Cache-Advisor stellt die quantitative Analyse des Füllverhaltens von Cache-Groups bzw. -Föderationen dar. Eine solche quantitative Analyse ermittelt die Kosten, die durch die Verwaltung und Instandhaltung einer Cache-Group entstehen.

Für eine quantitative Analyse gehen wir der Frage nach, wie eine kostenbasierte Bewertung aussehen kann und welche Einflussgrößen einzubeziehen sind. Wir wollen also zeigen, was wir unter Kosten verstehen und ein Kostenmodell entwickeln.

Das Caching eines großen Anteils der Daten aus dem Backend wird hauptsächlich dann problematisch, wenn die Daten im Backend oft aktualisiert werden. Denn dann muss der Cache ebenso diese Aktualisierung nachvollziehen. Dadurch erhöht sich die Netzwerklast, da die aktualisierten Sätze vom Backend zum Frontend übertragen werden müssen. Außerdem steigt die Systemlast im Frontend während der Aktualisierung, so dass sich die Antwortzeit erhöht.

Da im Frontend möglicherweise damit gerechnet wird, dass nur ein Teil der Daten vorgehalten werden muss, steht hier unter Umständen weniger Speicher zur Verfügung, und es kann durch die große Datenmenge zu einem Engpass der Ressource Hauptspeicher kommen. Das Ergebnis wäre z. B. ein Flattern (engl. thrashing) der Daten, da Seiten aus dem Hauptspeicher in den Hintergrundspeicher verdrängt werden, die anschließend aber sofort wieder zurück in den Hauptspeicher geholt werden, um eine Anfrage zu beantworten. Diese Situation würde in erheblichem Maße die Antwortzeit verschlechtern.

Aus den obigen Überlegungen ergibt sich, dass für ein schlechtes Performance-Verhalten des Caches zwei Faktoren entscheidend sind: Zum einen spielt die Anzahl n der zwischengespeicherten Daten und zum anderen die Häufigkeit a der Aktualisierung dieser Daten im Backend dabei eine Rolle. Wir gehen dabei von der Annahme aus, dass die Datensätze in verschiedenen Cache-Tabellen den gleichen Bedarf an Speicherplatz haben. Falls wir nicht von dieser Annahme ausgehen können (z. B. weil die Datensätze in einer bestimmten Tabelle sehr groß, dafür aber wenige sind), so benötigen wir Gewichte für die Cache-Tabellen. Der Einfachheit halber abstrahieren wir aber von einer unterschiedlichen Größe der Daten.

Werden die Daten im Backend so gut wie nie verändert, so entsteht durch das Caching dieser Daten kaum Aufwand für die Wartung und Replikation dieser Daten. Auf der anderen Seite führen häufige Aktualisierungen im Backend zu einer steigenden Netzwerk- und Systemlast, da die Aktualisierungen im Cache zeitnah nachvollzogen werden müssen. Betrachten wir die Häufigkeit der Aktualisierung, so können wir diese in verschiedenen Granularitäten auffassen.

- Zum einen können wir eine Aussage darüber treffen, wie häufig allgemein *Aktualisierungen in der gesamten Datenbank* durchgeführt werden. Diese Angabe wäre aber sicherlich nicht besonders aussagekräftig. Denn dann wäre dieser Faktor für alle möglichen Cache-Groups konstant, da die Tabellen im Frontend eine Teilmenge der Tabellen der gesamten Datenbank darstellen. Weil wir außerdem nichts an der Datenbank als solches ändern möchten, können wir auch hier keine Maßnahmen zur Performance-Verbesserung ansetzen.
- Im anderen Extremfall betrachten wir die Häufigkeit der *Aktualisierungen von einzelnen Sätzen*. Dieser Ansatz würde natürlich sehr genaue Daten liefern, welche Belegung einer Cache-Group zu einem schlechten Performance-Verhalten führen würde. Allerdings kann die Belegung einer Cache-Group zu verschiedenen Laufzeiten unterschiedlich sein, je nach dem, welche Cache-Keys vorgehalten werden sollen. Außerdem dürften statistische Daten hierüber nur mit erheblichem Aufwand verfügbar sein.
- Eine weitere Möglichkeit wäre die Betrachtung der Anzahl der *Aktualisierungen* innerhalb einer bestimmten Zeitspanne in den *Extensionen eines bestimmten Prädikatstyps*. Aufgrund der einfachen Verrechnung der Aktualisierungen pro Zeiteinheit mit der Anzahl in der entsprechenden Cache-Tabelle geladenen Sätze entscheiden wir uns gegen die Betrachtung von Prädikaten und für die Betrachtung einzelner Cache-Tabellen.
- Wir entscheiden uns also für einen vierten Ansatz: Bewertet wird die Anzahl der *Aktualisierungen* innerhalb einer bestimmten Zeitspanne *in einer Tabelle*.

Die Anzahl a_T der Aktualisierungen in einer Cache-Tabelle T pro Zeiteinheit müssen wir anhand der Beobachtung von vergangenen Aktualisierungen abschätzen. Wir nehmen dafür an, dass sich das Aktualisierungsverhalten in der Zukunft nicht maßgeblich ändern wird. Die Anzahl der Aktualisierungen während einer gerade abgelaufenen Zeitspanne wird dann als Schätzwert für die Anzahl der Aktualisierungen in einer gleich langen Zeitspanne in nächster Zukunft herangezogen.

Definition: Die Anzahl n_T der geladenen Sätze in einer Cache-Tabelle T heißt **Füllstand** dieser Cache-Tabelle.

Um ein Kriterium für die Güte einer Cache-Group im Sinne einer Kostenabschätzung zu erhalten, benötigen wir zunächst ein Maß für die Kosten, die das Caching einer Tabelle T in einer Cache-Group verursacht. Gemäß der oben identifizierten Probleme, lassen sich diese Kosten in zwei Kategorien aufteilen. Wir unterscheiden also zwei Kostenarten:

- **Kosten durch Bereitstellen von Speicherplatz** m_T und
- **Kosten durch steigende Netzwerk- und System-Last** l_T

Die Kosten m_T durch Bereitstellen von Speicherplatz für eine Cache-Tabelle T steigen dabei proportional zur Anzahl n_T der in die Cache-Tabelle T geladenen Sätze und sind unabhängig von der Anzahl a_T der Aktualisierungen der Tabelle innerhalb eine festen Zeitspanne:

$$m_T = n_T \cdot w_m$$

Das Gewicht w_m zeigt an, wie sehr der Speicherplatz einen Engpass darstellt und wie stark die Kosten dafür dementsprechend zu bewerten sind.

Die Kosten durch steigende Netzwerk- und System-Last steigen proportional sowohl zum Füllstand, als auch zur Anzahl der Aktualisierungen der Tabellen innerhalb einer gewissen Zeitspanne. Denn die Netzwerk- und System-Last steigt mit zunehmender Anzahl der Aktualisierungen, wobei nur die zwischengespeicherten Sätzen aktualisiert werden. Nicht berücksichtigt sind die Kosten der Prüfung, ob ein Satz überhaupt im Cache ist und somit aktualisiert werden muss. Wir gehen davon aus, dass diese Kosten konstant und unabhängig von Füllstand oder der Häufigkeit der Aktualisierungen sind.

$$l_T = n_T \cdot a_T \cdot w_l$$

Das Gewicht w_l zeigt an, wie sehr die Netzwerk- und System-Last einen Engpass darstellt und wie stark die Kosten dafür dementsprechend zu bewerten sind.

Addieren wir beide Komponenten m_T und l_T, so erhalten wir schließlich ein Maß für die gesamten Kosten, die das Caching einer Tabelle in einer Cache-Group verursacht.

Definition: Die durch das Zwischenspeichern der Cache-Tabelle T verursachten **Gesamtkosten** o_T entstehen durch Addition der Kosten durch Bereitstellen von Speicherplatz und durch steigende Netzwerk- und System-Last durch das Zwischenspeichern dieser Cache-Tabelle:

$$o_T = m_T + l_T$$

Um letztendlich ein Maß für die Kosten o_{CG}, die das Caching einer Cache-Group oder -Föderation CG verursacht werden, zu bilden, summieren wir die Gesamtkosten aller Cache-Tabellen T aus der Menge $\mathbf{CT}_{CG}$ der an CG beteiligten Cache-Tabellen:

$$o_{CG} = \sum_{T \in \mathbf{CT}_{CG}} o_T$$

Wir können das Maß für die Kosten verfeinern, indem wir die Gewichte nicht global setzen, sondern ebenfalls auf jeweils eine Cache-Tabelle bezogen. Dadurch lässt sich dann gegebenenfalls darstellen, wie stark der System-Engpass bezogen auf eine Cache-Tabelle ist.

Des Weiteren haben wir bisher nur die Kostenseite betrachtet. Immerhin soll durch das Caching eigentlich die Netzwerklast und die Systemlast im Backend reduziert werden. Wir müssten also noch eine Größe für den Nutzen der Cache-Group berechnen. Eine wichtige Einflussgröße hierfür ist, wie viele Anfragen aus dem Cache beantwortet werden können. Diese Frage hängt aber auch von Faktoren außerhalb des Designs der Cache-Group ab, z. B. der Art der Sondierung des Caches.

Außerdem hängt der Nutzen des Caches stark von der Referenzlokalität der Anfragen ab. Je häufiger Anfragen mit der gleichen Instanz eines parametrisierten Prädikats beim Cache ankommen, desto größer ist auch sein Nutzen, zumindest wenn dieser Typ von Prädikat durch die Definition der Constraints unterstützt wird. Wir müssen also für eine Nutzen-Analyse die ankommenden Anfragen beim Cache untersuchen und so feststellen, wie gut die unterstützten Prädikatstypen mit denen der ankommenden Anfragen überein stimmen. Wir werden uns in dieser Arbeit mit einer Analyse des Nutzens einer Cache-Group bzw. -Föderation nicht weiter beschäftigen.

Im Hauptteil dieser Arbeit wollen wir also das Füllverhalten von Cache-Groups modellieren. Die zu entwickelnden Modelle sollen dabei jeweils Abschätzungen der Füllstände in den Cache-Tabellen geben, nachdem ein einzelner Füllzyklus gemacht wurde. Da es durch die konkrete Wahl der referenzierten Cache-Keys und Cache-Key-Werte zu unterschiedlichen Füllständen kommen kann, sollen die Modelle durchschnittliche Werte liefern. Des Weiteren gehen wir davon aus, dass der Cache vor dem Füllzyklus leer war. Wir berechnen also die Anzahl der Sätze in den unterschiedlichen Cache-Tabellen, die durch dass Laden der Werteumgebung eines Cache-Key-Werts und der nachfolgenden Füllschritte in den Cache geladen werden müssen.

Wir werden für verschiedene Arten von Cache-Groups und -Föderationen Modelle für die Analyse ihres Füllverhaltens vorschlagen. Dabei kategorisieren wir die betrachtetes Cache-Groups wie folgt:

- Baumartige Cache-Groups haben an jeder ihrer Cache-Tabellen lediglich maximal eine eingehende RCC. Die Wurzel-Tabelle (die den Cache-Key enthält) besitzt keine eingehenden RCCs (wenn man von der Selbst-RCC für die Einstiegspunkteigenschaft des Cache-Keys absieht). Außerdem erlauben wir genau einen Cache-Key in der Cache-Group (in der Wurzel-Tabelle).
- RCC-zyklenfreie Cache-Groups enthalten keine RCC-Zyklen, so dass es insbesondere nicht zu einem rekursiven Laden kommen kann. Des Weiteren gestaltet sich die Berechnung der Füllstände noch relativ einfach, da wir die Cache-Tabellen in topologischer Sortierordnung betrachten können.
- Die dritte Klasse bilden allgemeine Cache-Groups und -Föderationen. Hier existieren keine Restriktionen bezüglich Zyklen oder der Anzahl der Cache-Keys.

Für baumartige Cache-Groups werden wir ein recht umfangreiches und genaues Modell vorschlagen, wobei hier auch Histogramme (das Wissen über die Häufigkeiten einzelner Werte in den Spalten der Cache-Tabellen) in die Berechnung der Füllstände einfließen. Für RCC-zyklenfreie Cache-Groups schlagen wir ein Modell vor, dass relativ einfach ist, dafür aber nicht so genau wie das Modell für baumartige Cache-Groups arbeitet. Für allgemeine Cache-Groups und -Föderationen gestaltet sich die Modellierung am schwierigsten. Wir schlagen zunächst ein Modell zur Worst-Case-Analyse vor, bei dem berechnet wird, wie viele Sätze maximal bei einem Füllzyklus in die einzelnen Cache-Tabellen geladen werden können. Außerdem zeigen wir einige Aspekte auf, die bei Inkaufnahme höherer Modellkomplexität und geringerer Genauigkeit Ansatzpunkte für eine Average-Case-Analyse bieten.

Wir werden des Weiteren das Modell zur Abschätzung der Füllstände in baumartigen Cache-Groups teilweise evaluieren. Die Ansätze zur Evaluierung können dabei prinzipiell auch für Modelle zur Analyse allgemeiner Cache-Groups eingesetzt werden, allerdings nur eingeschränkt.

2 Grundlagen

Nachdem wir recht ausführlich in die Thematik des DB-Caching und die Verwendung von Cache-Groups eingeführt haben, wollen wir nun den Fokus auf die Modellierung des Füllverhaltens von Cache-Groups richten. Zentral ist hierbei die Frage, wie viele Sätze in den verschiedenen Cache-Tabellen gespeichert sind, nachdem ein einzelner Füllzyklus bei einer zuvor leeren Cache-Group bzw. -Föderationen ausgeführt worden ist. Diese Größen (bezogen auf verschiedene Cache-Tabellen) nennen wir Füllstände. Bevor wir jedoch konkrete Modelle für verschiedene Spezialfälle von Cache-Groups entwickeln, wollen wir in diesem Kapitel allgemeine Grundlagen zur Entwicklung dieser Modelle betrachten.

Dieser Kapitel ist wie folgt aufgebaut:

In Abschnitt 2.1 stellen wir Modellannahmen vor, die wir zur quantitativen Analyse des Füllverhaltens von Cache-Groups benötigen.

Anschließend betrachten wir in Abschnitt 2.2 den so genannten Spalten-Beziehungsgraph, der RCCs und Schmuggler-Beziehungen zwischen Spalten darstellt. Aufgrund dieser Struktur können wir die Spalten in einer Cache-Group kategorisieren und ihre Eigenschaften untersuchen.

In Abschnitt 2.3 untersuchen wir Füllschritte und betrachten diese auf verschiedenen Granularitätsebenen. D. h. wir unterscheiden die Füllschritte je nach dem, wie viele Werte bei einem betrachteten Füllschritt einstiegsfähig gemacht werden. Außerdem betrachten wir in diesem Abschnitt verschiedene Zeitpunkte der Verarbeitung, die in einem Algorithmus zur Bewertung des Füllverhaltens modelliert werden.

In Abschnitt 2.4 sind Werte und Wertemengen die betrachteten Objekte. Diese spielen eine zentrale Rolle bei der Bewertung des Füllverhaltens von Cache-Groups. Je mehr Werte z. B. in den Spalten der Cache-Tabellen geladen sind, desto größer sind im Allgemeinen auch die Füllstände. Die Betrachtungen in diesem Abschnitt sollen uns einen Überblick über die verschiedenen Zusammenhänge zwischen Wertemengen bieten.

In einem Exkurs beschäftigen wir uns danach in Abschnitt 2.5 mit der Abschätzung der Mächtigkeit einer Schnittmenge zweier Grundmengen. Dies ist notwendig, da in allen später entwickelnden Modellen diese Frage in der einen oder anderen Situation beantwortet werden muss, um die Füllstände abzuschätzen.

In Abschnitt 2.6 beschäftigen wir uns wird Histogrammen und der Zugehörigkeitsfunktion. Histogramme werden hauptsächlich zur Analyse baumartiger Cache-Groups.

Im anschließenden Abschnitt 2.7 betrachten wir das Verhalten von Cache-Keys und gehen der Frage nach, wie viele Sätze beim ersten Füllschritt einem Füllzyklus in eine Cache-Group geladen werden.

Danach betrachten wir in Abschnitt 2.8 RCCs und deren Auswirkung auf das Füllverhalten. Zunächst betrachten wir die RCC-Eigenschaft formal und analysieren, welche Auswirkungen die Eigenschaft hat. Anschließend nehmen wir die so genannte RCC-Teilmengen-Annahme unter die Lupe, die besagt, dass zwei durch eine RCC verbundene Spalten eine Teilmengen-Beziehung haben. Danach schauen wir uns die Überdeckung zwischen zwei Spalten, die durch eine RCC verbunden sind. Die Überdeckung gibt dabei das Verhältnis von der Mächtigkeit der Schnittmenge dieser beiden Spalten zur Mächtigkeit der DBD des Konsumenten an und hilft uns bei der Abschätzung der Anzahl der während eines Füllschritts in den Konsumenten induzierten Werte. Gewissermaßen eine transitive Fortsetzung von RCCs für einzelne Werte stellen die ebenfalls in diesem Abschnitt betrachteten Wertemengen-Pfade dar.

In Abschnitt 2.9 schauen wir uns Schmuggler-Beziehungen an und schätzen die Anzahl der Werte ab, die bei einem Füllschritt in ein Schmuggelziel eingeführt werden. Außerdem berechnen wir die Wahrscheinlichkeit, mit der ein Wert in ein Schmuggelziel eingeführt wird.

Abschließend geben wir in Abschnitt 2.10 eine Zusammenfassung des in diesem Kapitel 2 gewonnenen Wissens.

2.1 Modellannahmen

In den Algorithmen zur quantitativen Analyse des Füllverhaltens in einer Cache-Group oder Föderation verwenden wir immer wieder Annahmen zur Vereinfachung der Berechnungen. Einige dieser Modellannahmen wollen wir in diesem Abschnitt definieren und unter die Lupe nehmen.

Um die Anzahl der für einen Wert geladenen Sätze abschätzen zu können, benötigen wir Informationen über die Verteilung der Werte in den Spalten. Im Allgemeinen treffen wir in dieser Arbeit die Annahme, dass die Anzahl der Sätze, die einen bestimmten Wert in einer Spalte haben, gleich groß für jeden konkreten Wert aus der DBD einer Spalte ist. Die Werte sind also statistisch gleich über den Wertebereich einer jeweiligen Spalte verteilt.

Definition: Die **Gleichverteilungsannahme** besagt, dass die Werte in den Spalten im Backend über den Wertebereich gleichverteilt sind.

Können wir nicht von der Gleichverteilungsannahme ausgehen, so benötigen wir Histogramme, die Auskunft darüber geben, wie häufig ein Wert in der Spalte vorkommt.

Um Schmuggler-Beziehungen zu analysieren, müssen wir wissen, ob es eine Beziehung zwischen Werten in den Spalten am Anfang und am Ende der Schmuggler-Beziehung gibt.

Definition: Die **Spaltenunabhängigkeitsannahme** besagt, dass die Verteilung der Werte in verschiedenen Spalten stochastisch unabhängig ist.

Es existiert also keine Korrelation zwischen Werten in den Spalten am Anfang und Ende einer Schmuggler-Beziehung.

Können wir nicht von der Spaltenunabhängigkeitsannahme ausgehen, so benötigen wir einen Korrelationskoeffizienten. Hierdurch würde die Abschätzung der Anzahl der in eine Spalte geschmuggelten Werte zwar verbessert, aber dies wäre mit einem ungerechtfertigten Mehr-

aufwand verbunden. Aus diesem Grund gehen wir bei der Modellierung der Cache-Groups und -Föderationen grundsätzlich von der Spaltenunabhängigkeitsannahme aus.

Auch die Betrachtung von Null-Werten würde die Modellierung enorm erschweren. Aus diesem Grund gehen wir grundsätzlich davon aus, dass keine Null-Werte in den Spalten vorkommen. Man beachte, dass mit Null-Werten nicht die Zahl „0“ gemeint ist, sondern dieser Satz in der Spalte keinen Wert hat.

Es handelt sich bei der Gleichverteilungsannahme, der Spaltenunabhängigkeitsannahme und der Annahme, dass keine Null-Werte in den Spalten existieren um die üblichen Annahmen, die bei einer Bewertung von Datenbank-Relationen z. B. bei der Erstellung von Zugriffsplänen bei Joins, getroffen werden [Här03].

2.2 Der Spalten-Beziehungsgraph

Nachdem wir nun grundlegende Begriffe definiert haben, betrachten wir in diesem Abschnitt die benötigte Struktur (ein Graph), um dass Füllverhalten einer Cache-Group modellieren zu können. Dieser so genannte Spalten-Beziehungsgraph repräsentiert die wichtigsten Elemente der Cache-Groups, d. h. die Spalten. Außerdem werden die Zusammenhänge zwischen den Spalten in der Struktur repräsentiert. Im Gegensatz zu einem einfachen **Erreichbarkeitsgraph**, der lediglich Cache-Tabellen durch Knoten und RCCs durch dazwischen liegende gerichtete Kanten darstellt [Här03], erhalten wir mit dem Spalten-Beziehungsgraph auch direkt Informationen über die Beziehungen der Spalten innerhalb einer Cache-Tabelle. Dieses Mehr an Information wird von den Algorithmen zur Berechnung der Füllstände einer Cache-Groups oder -Föderation aufgegriffen.

Definitionen:

- Ein **Spalten-Beziehungsgraph (SBG)** ist ein zusammenhängender Graph, bei dem die betrachteten Spalten als Knoten dargestellt sind, die durch gerichtete Kanten verbunden sind, wenn zwischen ihnen eine RCC oder Schmuggler-Beziehung besteht.[9]
- Zwischen den Knoten eines SBG befinden sich zwei Arten von gerichteten Kanten: **RCC-Kanten** und **Schmuggler-Kanten**.
- Bei den Knoten in einem SBG handelt es sich nur um **relevante Spalten**. Als **relevant** bezeichnen wir diejenigen Spalten, bei denen entweder RCCs beginnen und/oder enden, zuzüglich der Cache-Keys.
- Bei einer RCC e→k nennen wir k den **Nachfolger** von e. Umgekehrt wird e der **Vorgänger** von k genannt. Diese Bezeichnungen verwenden wir allerdings nicht bei Spalten am Anfang bzw. Ende einer Schmuggler-Kante.

Spalten, welche nicht relevant sind, werden für die Modellierung des Füllverhaltens nicht benötigt, da sie das Füllverhalten einer Cache-Group nicht beeinflussen.

[9] In einem SBG existieren keine Schmuggler-Kanten, die eine Spalte mit sich selbst verbinden, denn eine Spalte kann niemals in einer Schmuggler-Beziehung zu sich selbst stehen.

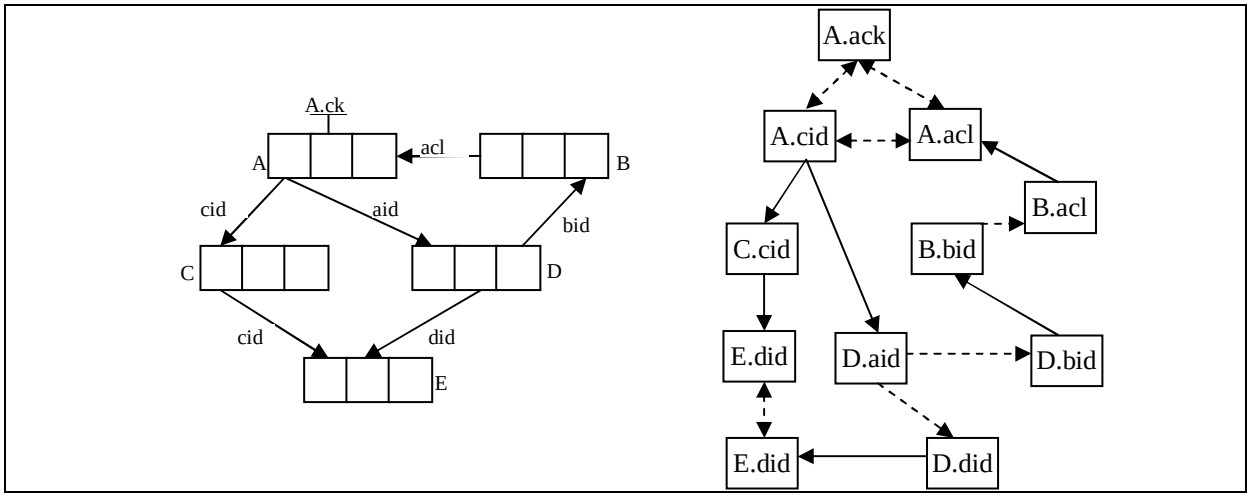

Abbildung 4: Eine Cache-Group als Erreichbarkeitsgraph und als SBG

Beispiel: Abbildung 4 zeigt auf der linken Seite eine Cache-Group als Erreichbarkeitsgraph. Die Beschriftung an den RCCs geben den Namen der Spalte am Anfang und Ende der RCC wieder. Die Namen stimmen also im abgebildeten Beispiel überein. Auf der rechten Seite dieser Abbildung ist die gleiche Cache-Group als SBG zu sehen. Die Knoten stellen die Spalten der Cache-Group dar. Zwischen den Knoten befinden sich gerichtete Kanten, wobei die Kanten mit Pfeilen an beiden Enden für zwei Kanten jeweils in eine Richtung und die umgekehrte Richtung stehen. RCC-Kanten werden durch durchgezogene Pfeile und Schmuggler-Kanten werden durch gestrichelte Pfeile dargestellt.

Im Rest des Abschnitts wollen wir mögliche Zerlegungen der Menge der Spalten (im SBG) einer Cache-Group beleuchten und dabei wichtige Begriffe bilden, die uns bei der Beschreibung der Zusammenhänge in unseren Modellen helfen.

Diese Spalten stellen die Knoten im SBG dar. Wir wollen die Spalten einer Cache-Group also verschiedenen Kategorien zuordnen. Durch diese Kategorisierung erhalten wir wichtige Begriffe, die wir bei der späteren Modellierung der Cache-Groups und -Föderationen benötigen.

Eine mögliche Zerlegung erhält man, indem man die Spalten betrachtet, die zur selben Cache-Tabelle gehören. Die Menge der Knoten des SBG lässt sich so nach der Zugehörigkeit der Spalten zu verschiedenen Tabellen partitionieren.

Weiterhin können wir Spalten aber auch danach unterscheiden, ob bei ihnen RCCs beginnen oder enden.

Definitionen:

- Eine Spalte heißt **Konsument**, wenn an ihr eine RCC-Kante endet. Wenn an einer Spalte eine RCC-Kante anfängt, so wird diese Spalte **Erzeuger** genannt.[10]

[10] Eine Spalte kann natürlich sowohl Konsument als auch Erzeuger sein: An einer solchen Spalte endet eine RCC und beginnt eine (eventuell andere) RCC.

- Eine Spalte heißt **Schmuggelquelle**, wenn an ihr eine Schmuggler-Kante beginnt. Wenn an einer Spalte eine Schmuggler-Kante endet, so wird diese Spalte **Schmuggelziel** genannt.
- Wir bezeichnen eine Spalte, die ein Erzeuger ist, aber kein Konsument, als **Basiserzeuger**.[11]

Basiserzeuger haben die Eigenschaft, dass in sie keine Werte induziert werden können. Dementsprechend sind alle geladenen Werte in eine solche Spalte geschmuggelt worden, wenn es sich nicht um einen Cache-Key handelt.

Den Begriff „Konsument“ können wir in Anlehnung an ein Erzeuger-Verbraucher-Paradigma erklären. Bei Systemen, in denen Aktionen asynchron ausgeführt werden, ist eine mögliche Verarbeitung der Aktionen dadurch gekennzeichnet, dass Abhängigkeiten zwischen den Objekten, die die Aktionen ausführen, bestehen. Wenn ein Objekt eine Aktion ausführen kann, so müssen Objekte, von denen das ausführende Objekt abhängt, vorher bereit sein. Diese Bereitschaft wird durch Token repräsentiert, die durch Objekte am Anfang der Abhängigkeit erzeugt werden und die durch Objekte am Ende der Abhängigkeit durch die Ausführung der Aktion verbraucht werden. Man sagt dann auch, das Token wird konsumiert. Im Zusammenhang mit Cache-Groups ergibt sich der Konsum durch die Verarbeitung ankommender Werte. Die zugehörige Aktion ist dann ein Füllschritt.

Im Rahmen dieser Arbeit sagen wir entsprechend auch, dass ankommende Werte konsumiert werden. Man beachte allerdings, dass die Werte nicht einfach verschwinden, sondern durchaus über RCCs bei Nachfolgern ankommen können.

2.3 Füllschritte

Grundlegende Ereignisse beim Füllverhalten einer Cache-Group oder -Föderation stellen Füllschritte dar. Während eines Füllschritts werden Sätze in eine Cache-Tabelle geladen und dadurch neue Werte in die relevanten Spalten der Cache-Tabelle eingeführt. Das Laden neuer Sätze in die Cache-Tabelle beeinflusst direkt den Füllstand einer Cache-Tabelle, also die zur Analyse des Füllverhaltens einer Cache-Group bzw. -Föderation in dieser Arbeit hauptsächlich betrachteten Größe. Alle Algorithmen zur Berechnung der Füllstände in Cache-Groups oder -Föderationen müssen also Füllschritte betrachten.

Ein Füllschritt wird prinzipiell dadurch ausgelöst, dass eine Spalte der Cache-Tabelle ungesättigt ist.

Wird ein Füllschritt ausgeführt, weil eine Spalte in der Cache-Tabelle Werte aus ihrer DBD nicht enthält, die in einem Vorgänger geladen sind, so müssen Sätze aus dem Backend geladen werden, um die RCC gültig zu machen. Nach dem Füllschritt sind einige dieser Werte in der betrachteten Spalte umgebungsvollständig, und diejenigen, die es danach nicht sind, müssen in weiteren Füllschritten noch umgebungsvollständig gemacht werden, um die Cache-

[11] Es handelt sich dabei um Erzeuger an der Basis (d. h. am Anfang) eines zusammenhängenden Wegs aus RCC-Kanten.

Group bzw. -Föderation in einen gültigen Zustand zu versetzen. Durch das Laden von Sätzen werden allerdings in andere Schmuggelziele ebenfalls Werte eingeführt (geschmuggelt).

Durch das Schmuggeln von Werten in die anderen Spalten (insbesondere in die Erzeuger) der Cache-Tabelle werden die Nachfolger der Erzeuger eventuell ungesättigt. Die geschmuggelten Werte kommen also beim Konsumenten der nachfolgenden Cache-Tabelle an usw.

Wir können einen Füllschritt also in zwei Wirkzusammenhänge unterteilen.

- **Wirkzusammenhang I** beschreibt das Laden von Werten, die über eingehende RCCs ankommen.
- **Wirkzusammenhang II** beschreibt das Schmuggeln von Werten in die Schmuggelziele der Cache-Tabelle während eines Füllschritts.

Beim Wirkzusammenhang I stehen also die RCCs im Mittelpunkt. Durch das Laden der ankommenden Werte werden Sätze in die Cache-Tabelle geladen, welche verursachen, dass in die Erzeuger Werte geschmuggelt werden. Im Mittelpunkt von Wirkzusammenhang II steht die Schmuggler-Beziehung.

In Abbildung 5 ist ein Beispiel für das Zusammenspiel der Wirkzusammenhänge dargestellt. Wir betrachten drei Spalten j, k und e. Die Spalten k und e gehören zur gleichen Cache-Tabelle, wobei k eine Spalte mit eingehender RCC und e eine Spalte mit ausgehender RCC der Cache-Tabelle ist. Die Spalte j ist der Vorgänger von k. Wir betrachten also die RCC $j \rightarrow k$, die den Wirkzusammenhang I bestimmt und $k \rightsquigarrow e$, wodurch der Wirkzusammenhang II bestimmt wird.

Beide Wirkzusammenhänge stehen in Verbindung mit der Anzahl der in die Cache-Tabelle geladen Sätze. Diese Anzahl wird durch den Wirkzusammenhang I bestimmt und beeinflusst den Wirkzusammenhang II.

Allgemein gibt es bei einem Füllschritt die Wirkzusammenhänge I und II mehrmals, da in einer Cache-Tabelle prinzipiell mehrere Konsumenten und Schmuggelziele existieren. So gibt es dementsprechend auch mehrere Schmuggler-Beziehungen zwischen jeweils zwei Spalten und damit auch mehrere Wirkzusammenhänge II. Auf der anderen Seite können wir für jede Spalte mit eingehender RCC einen Wirkzusammenhang I betrachten. Auch dieser Wirkzusammenhang besteht dementsprechend mehrfach.

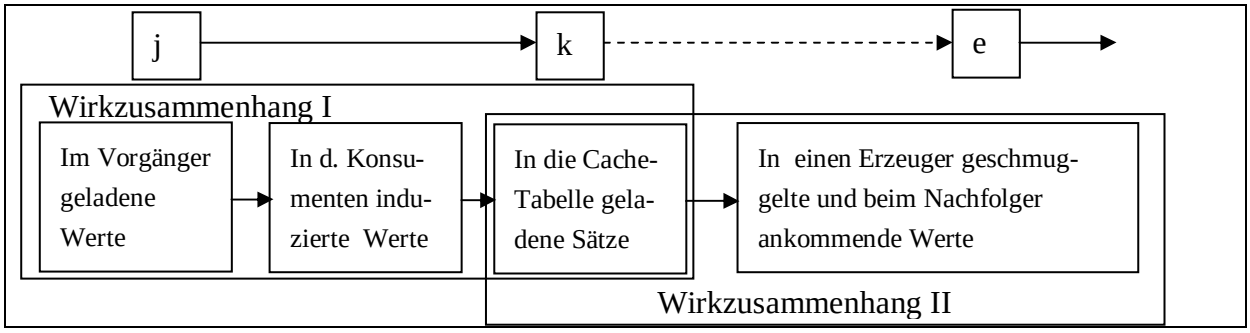

Abbildung 5: Zusammenspiel der Wirkzusammenhänge

Wird ein Füllschritt dadurch verursacht, dass ein Cache-Key-Wert durch eine Benutzeranfrage referenziert wird, so ist nicht eine RCC die maßgebliche Einflussgröße auf die geladenen

Sätze, sondern der Cache-Key. Alle Sätze, die den referenzierten Cache-Key-Wert im Cache-Key haben, werden in die Tabelle geladen. Diese verursachen allerdings wie gehabt im Rahmen des Wirkzusammenhangs II das Schmuggeln von Werten in Schmuggelziele.

Füllschritte auf verschiedenen Granularitätsebenen

Je nach dem, welche Granularität wir für die Füllschritte zu Grunde legen, werden unterschiedliche Mengen von ankommenden Werten nach dem Füllschritt umgebungsvollständig sein. Die Algorithmen, die auf die verschiedenen Arten von Füllschritten zurückgreifen haben dementsprechend mehr oder weniger Informationen zur Verfügung.

Die Granularität unterscheidet sich in der Mächtigkeit der Satzmengen, die während eines Füllschritts geladen werden. Wir unterscheiden je nach betrachteter Granularität die folgenden Arten von Füllschritten

- **Einzelwert-Füllschritte** machen jeweils einen einzelnen Wert, der in einem Vorgänger geladen ist, einstiegsfähig. Ist der betrachtete Wert in der DBD der aktiven Spalte vorhanden, so ist er nach dem Füllschritt umgebungsvollständig.
- **Direkte RCC-Füllschritte** werden potentiell immer dann ausgeführt, wenn sich die Menge der geladenen Werte im Vorgänger verändert und dabei neue Werte hinzukommen, die in der DBD des Konsumenten vorhanden aber dort nicht umgebungsvollständig sind. Direkte RCC-Füllschritte müssen dabei nicht alle ankommenden Werte einstiegsfähig machen. Welche und wie viele der Werte einstiegsfähig gemacht werden, hängt vom betrachteten Algorithmus ab. Es muss lediglich mindestens einer der Werte während des Füllschritts induziert werden (ansonsten könnten wir von diesem Füllschritt abstrahieren, denn er verändert den Zustand der Cache-Groups nicht).
- Im Gegensatz zu direkten RCC-Füllschritten werden **zusammenfassende RCC-Füllschritte** generell erst dann ausgeführt, wenn endgültig fest steht, wie viele Werte in den Vorgänger am Anfang der RCC geladen wurden. Dessen Cache-Tabelle muss also zunächst alle Füllschritte ausgeführt haben (d. h. vollständig bevölkert sein, siehe Abschnitt 1.7), bevor die nachfolgende Cache-Tabelle einen RCC-Füllschritt ausführen kann. Nach dem Füllschritt sind alle über die betrachtete RCC ankommenden Werte in der aktiven Spalte einstiegsfähig. Die Werte unter diesen, die in der DBD der aktiven Spalte vorhanden sind, sind nach dem Füllschritt umgebungsvollständig.

Prinzipiell sind weitere Arten von Füllschritten durchaus denkbar. In realen Systemen zum Datenbank-Caching mit Cache-Groups werden wahrscheinlich alle Sätze, die in beliebigen Vorgängern einer Cache-Tabelle geladen und in ihren korrespondierenden Konsumenten noch nicht umgebungsvollständig sind auf einmal induziert; Zumindest ist dieses Verhalten denkbar. Für die in dieser Arbeit im Mittelpunkt stehenden Algorithmen zur Berechnung der Füllstände in einer Cache-Group spielen andere als die vorgestellten Arten von Füllschritten allerdings keine Rolle.

Wir nehmen im Rahmen dieser Arbeit an, dass ein Algorithmus zur Berechnung der Füllstände Füllschritte für jede Cache-Tabelle auf der gleichen Granularitätsebene betrachtet. Prinzipiell kann man sagen, dass die Abschätzung genauer wird, je feiner die Granularität ist. Denn dann steht einem Algorithmus im Allgemeinen ein Mehr an Information über die Wertemen-

gen, die während eines jeweiligen Füllschritts umgebungsvollständig gemacht werden, zur Verfügung.

Bei Einzelwert-Füllschritten ist die aktive Spalte (d. h. die Spalte, in die Werte induziert werden, weil sie ungesättigt ist; siehe Abschnitt 1.7) diejenige, in die der betrachtete Wert geladen wird. Betrachten wir RCC-Füllschritte, so ist die aktive Spalte die Spalte am Ende der betrachteten RCC.

Eine Abschätzung der Füllstände unter Zuhilfenahme von Einzelwert-Füllschritten führt im Allgemeinen zu einer verlängerten Laufzeit des Algorithmus, da jeder Wert einzeln angefasst werden muss (für jeden Wert aus der DBD einer Spalte wird von einem solchen Algorithmus ein Füllschritt betrachtet).

Positiv wirkt sich allerdings aus, dass mit Hilfe von Einzelwert-Füllschritten Histogramme in die Berechnung der Füllstände einfließen können, um die Genauigkeit der Abschätzung zu verbessern. Aus diesem Grund werden Einzelwert-Füllschritt im Rahmen dieser Arbeit lediglich in Algorithmen zur Abschätzung der Füllstände von baumartigen Cache-Groups verwendet.

Wie wir gesehen haben, stellen Füllschritte ein wichtiges Element für die Betrachtungen des Füllverhaltens einer Cache-Groups oder -Föderation dar. Um die Cache-Group nach einem Cache-Miss zu füllen, müssen im Allgemeinen mehrere Füllschritte ausgeführt werden – zum einen, um die Extension des Prädikats der Benutzeranfrage in den Cache zu holen, zum anderen, um die Constraints (Cache-Key und RCC-Eigenschaft) durchzusetzen und den Cache somit wieder in einen gültigen Zustand zu versetzen.

Zeitpunkte der Verarbeitung

Die Füllschritte in einem Füllzyklus geschehen im Allgemeinen zu unterschiedlichen Zeitpunkten. Da alle Algorithmen zur Berechnung der Füllstände in einer Cache-Group oder -Föderation Füllschritte betrachten, ist auch hier die Anzahl und Reihenfolge der Füllschritte wichtig.

Wir wollen also die verschiedenen (logischen) Zeitpunkte untersuchen, um Abhängigkeiten zwischen den Füllschritten zu analysieren und einen Anhaltspunkt für die Laufzeit der Algorithmen zur Modellierung des Füllverhaltens zu bekommen: je größer die Anzahl der zu betrachteten Zeitpunkte und damit Füllschritte ist, desto länger braucht der Algorithmus im allgemeinen zur Berechnungen der abgeschätzten Füllstände.

Wir haben dabei keinen kontinuierlichen Zeitbegriff, sondern unterscheiden nur diskrete Zeitpunkte. Bei diesen Zeitpunkten handelt es sich dabei um diejenigen, zu denen ein Füllschritt ausgeführt wird, sowie um zwei zusätzliche Zeitpunkte α und ω vor bzw. nach allen betrachteten Füllschritten. Andere Zeitpunkte müssen nicht betrachtet werden, da sie für die Modellierung des Füllverhaltens nicht relevant sind. Der in diesem Grundlagenteil und in den anschließenden Modellen betrachtete Ablauf beginnt grundsätzlich mit dem Referenzieren eines Cache-Key-Werts in einer Benutzeranfrage. Danach wird die Extension des Anfrageprädikats der Benutzeranfrage in den Cache geladen. Um die durch die Cache-Group definierten Constraints zu erfüllen, müssen eventuell weitere Sätze in den Cache geladen werden. Im ersten Füllschritt werden dementsprechend die Sätze, die den referenzierten Cache-Key-Wert

im Cache-Key enthalten, geladen. Anschließend werden so lange weitere Sätze geladen, bis der Cache wieder einen gültigen Zustand hat.

Man beachte, dass die Menge aller Zeitpunkte F immer endlich ist. Wir können also sogar bei einem rekursiven Laden davon ausgehen, dass der Ladevorgang irgendwann abgeschlossen ist. Es werden spätestens dann keine Füllschritte mehr ausgeführt, wenn alle im Backend vorhandenen Sätze in den jeweiligen Cache-Tabellen geladen sind.

Bei den Füllschritten handelt es sich prinzipiell um nebenläufige Prozesse, d. h. die Mengen F_T der Zeitpunkte, zu denen eine einzelne Cache-Tabelle T Füllschritte ausführt, sind für verschiedene Cache-Tabellen $T \in CG$ einer Cache-Group bzw. -Föderation CG allgemein nicht disjunkt. Immer wenn die Menge der geladenen Werte in einer Spalte am Anfang einer RCC keine Teilmenge der Menge der einstiegsfähigen Werte in der Spalte am Ende einer RCC ist, d. h. wenn die RCC-Eigenschaft für eine RCC nicht erfüllt ist, kann durch die Cache-Tabelle am Ende der RCC ein Füllschritt ausgeführt werden. Zu einem gegebenem Zeitpunkt kann es natürlich sein, dass für mehrere RCCs die RCC-Eigenschaft nicht erfüllt ist. In diesem Fall kann von mehreren Cache-Tabellen ein Füllschritt ausgeführt werden. Diese Füllschritte können zu unterschiedlichen aber auch zum gleichen Zeitpunkt ausgeführt werden.

Im Rahmen unserer Betrachtungen gehen wir der Einfachheit halber davon aus, dass zu jedem Zeitpunkt maximal ein Füllschritt ausgeführt wird. In einer realen Implementierung kann dies natürlich anders aussehen. Wir abstrahieren in unseren Modellen allerdings von dieser Möglichkeit, und gehen davon aus, dass die Füllschritte in einer gewählten Reihenfolge, einer so genannten **Serialisierung**, ausgeführt werden. Der Begriff kommt daher, dass die Füllschritte hintereinander, also seriell, ausgeführt werden.

Je nach Serialisierung der Füllschritte ergeben sich natürlich unterschiedliche Mengen F_T und je nach Algorithmus kann die Serialisierung unterschiedlich aussehen: Denn wenn mehrere Cache-Tabellen einen Füllschritt machen können, so hängt es vom jeweiligen Algorithmus ab, welche Cache-Tabelle gewählt wird.

2.4 Werte und Wertemengen

Werte und Wertemengen spielen eine zentrale Rolle bei der Bewertung des Füllverhaltens von Cache-Groups. Je mehr Werte z. B. in den Spalten der Cache-Tabellen geladen sind, desto größer sind im Allgemeinen auch die Füllstände. Die Betrachtungen in diesem Abschnitt sollen uns einen Überblick über die verschiedenen Zusammenhänge zwischen Wertemengen geben. Wir werden im einzelnen die DBD einer Spalte partitionieren, was uns später die Abschätzung der Anzahl geladener Sätze ermöglicht. Außerdem werden wir die Aufteilung der geladenen Werte in induzierte und geschmuggelte Werte betrachten.

Wir bezeichnen Wertemengen im Folgenden mit einem großen V und einem entsprechenden Index. Die abgeschätzte Mächtigkeit einer Wertemenge notieren wir mit einem kleinen v und dem gleichen Index. Für eine Übersicht der in dieser Arbeit verwendeten Wertemengen und ihrer Indizes sei auf Anhang B verwiesen. Dort werden Schreibweisen erläutert und ein Symbolverzeichnis aufgeführt.

Während eines Füllschritts gelangen neue Werte in die verschiedenen Spalten einer Cache-Tabelle. Ein Wert φ heißt zum Zeitpunkt τ in Spalte c **zusätzlich geladen** (bzw. **zusätzlich umgebungsvollständig**, **zusätzlich induziert**, **zusätzlich geschmuggelt**), wenn φ vor τ noch nicht in c geladen (bzw. umgebungsvollständig, induziert, geschmuggelt) war, es aber unmittelbar danach ist.

Die Menge der während eines Zeitpunkts zusätzlich geladenen Werte ist wichtig für die Modellierung allgemeiner Cache-Groups und -Föderationen. Hier kann mit Hilfe der zusätzlich induzierten Werte die Anzahl der während eines Füllschritts zu ladenden Sätze abgeschätzt werden. Ebenso kann mit Hilfe der zusätzlich in die Erzeuger einer Cache-Tabelle geschmuggelten Werte abgeschätzt werden, wie viele Werte ein Laden von Sätzen in ihren Nachfolgern verursachen.

Die DBD V_c einer Spalte c kann zu jedem Zeitpunkt τ in drei Partitionen zerlegt werden: Die Mengen der umgebungsvollständigen, umgebungsfragmentarischen und ungeladenen Werte. Diese Mengen sind paarweise disjunkt und die Vereinigung der Mengen ist die Menge V_c. Ob ein Wert in einer Spalte umgebungsvollständig, umgebungsfragmentarisch oder ungeladen ist, hat einen direkten Einfluss auf die in der Cache-Tabelle geladen Sätze. Prinzipiell sind für einen umgebungsvollständigen Wert mehr Sätze geladen, als wenn dieser Wert lediglich umgebungsfragmentarisch wäre. Für einen ungeladenen Wert sind natürlich keinerlei Sätze in der Cache-Tabelle gespeichert.

Die erste Teilmenge $V_{vc|c|\tau}$ bilden die Werte, die zum Zeitpunkt τ umgebungsvollständig im Cache vorhanden sind. Alle Sätze, die diesen Wert in der betrachteten Spalte haben, sind zum betrachteten Zeitpunkt im Cache geladen.

Eine andere Partition $V_{vf|c|\tau}$ von V_c bilden diejenigen Werte, von denen zum Zeitpunkt τ lediglich eine echte Teilmenge der Werteumgebung im Cache vorhanden sind. Eine solche Situation entsteht dann, wenn ein Wert in den Konsumenten geschmuggelt wird. Wir bezeichnen einen Wert, für den mindestens einer aber nicht alle der Sätze aus seiner Werteumgebung geladen sind, als **umgebungsfragmentarisch**. Den Begriff können wir uns anschaulich vorstellen, indem wir uns vor Augen halten, dass nur ein Fragment der Werteumgebung eines umgebungsfragmentarischen Werts in der Cache-Tabelle geladen ist.

Die dritte Partition $V_{n|c|\tau}$ von V_c bilden die Werte, von denen zum Zeitpunkt τ noch keiner der Sätze aus seiner Werteumgebung im Cache vorhanden ist. Die Werte wurden also weder in den Konsumenten geschmuggelt, noch sind sie z. B. durch Induzieren umgebungsvollständig.

Da die drei Mengen $V_{vc|c|\tau}$, $V_{vf|c|\tau}$ und $V_{n|c|\tau}$ eine Partitionierung der DBD der Spalte c aus der Menge **COL** einem bestimmten Zeitpunkt τ darstellen und die Teilmengen somit disjunkt sind, können wir für die Mächtigkeiten $v_{vc|c\tau}$, $v_{vf|c\tau}$ und $v_{n|c\tau}$ der Partitionen und der Kardinalität c_c der Spalte c folgenden Zusammenhang aufzeigen:

$$c_c = v_{vc|c|\tau} + v_{vf|c|\tau} + v_{n|c|\tau} \text{ für alle } \tau \in F,\ c \in \mathbf{COL} \qquad (2.4.1)$$

Grundsätzlich werden Werte im Rahmen von Wirkzusammenhang I in eine Spalte induziert. Insbesondere muss eine Spalte ein Konsument oder ein Cache-Key sein, damit Werte in sie induziert werden können. Geschmuggelte Werte gelangen prinzipiell im Rahmen von Wirkzusammenhang II in ein Schmuggelziel. Hierbei gibt es allerdings keine Einschränkung, dass die Spalte ein Konsument oder Erzeuger sein muss.

Man beachte, dass die Menge $\tilde{V}_{s|c|\tau}$ der während eines Füllschritts geschmuggelten und die Menge $V'_{s|c|\tau}$ der währenddessen *zusätzlich* geschmuggelten Werte nicht identisch ist. So können während eines Füllschritts über Wirkzusammenhang II Werte in die Spalte geschmuggelt werden, die während eines vorhergehenden Füllschritts bereits in die Spalte geschmuggelt wurden. Für induzierte Werte φ ($\in V'_{s|c|\tau}$) gilt dies prinzipiell aber nicht, da nur Füllschritte für solche Werte ausgeführt werden, die vorher noch nicht induziert waren.

Jeder induzierte Wert ist natürlich auch umgebungsvollständig:

$$V_{i|k|\tau} \subseteq V_{vc|k|\tau} \text{ für alle } \tau \in F, k \in \mathbf{COL} \qquad (2.4.2)$$

Die Menge der umgebungsfragmentarischen Werte ist eine Teilmenge der Menge der geschmuggelten Werte:

$$V_{vf|c|\tau} \subseteq V_{s|c|\tau} \text{ für alle } \tau \in F, c \in \mathbf{COL}. \qquad (2.4.3)$$

Ein Wert kann nur dann umgebungsfragmentarisch sein, wenn er in die Spalte geschmuggelt wurde. Induzierte Werte sind immer umgebungsvollständig, damit die RCC-Eigenschaft erfüllt ist. Ein Wert kann aber nur durch Induzieren oder durch Schmuggeln in eine Spalte gelangen.

Man beachte, dass es sich bei der oben dargestellten Beziehung im Allgemeinen um eine echte Teilmengen-Beziehung handelt. D. h. nicht alle geschmuggelten Werte sind umgebungsfragmentarisch. Zum einen kann ein geschmuggelter Wert auch induziert worden sein, wodurch er umgebungsvollständig ist. Zum anderen gibt es Fälle, in denen kein geschmuggelter Wert umgebungsfragmentarisch sein kann: In Spalten, die unique sind, d. h. deren Kardinalität der Kardinalität der Cache-Tabelle entspricht, entstehen durch Schmuggeln keine umgebungsfragmentarischen Werte. Da jeder Satz in der Cache-Tabelle einen einzigartigen Wert in diesem Konsumenten enthält, ist der referenzierte Wert per Definition umgebungsvollständig.

Bemerkung: Bei baumartigen Cache-Groups (d. h. Cache-Groups, deren SBG ein Baum und deren einziger Cache-Key der Wurzel-Knoten ist), wie sie in Kapitel 3 vorausgesetzt werden, kann auf die Modellierung der Mengen $V_{vf|c|\tau}$ und $V_{s|c|\tau}$ verzichtet werden kann. Diese sind zwar prinzipiell vorhanden (wie bei jeder Cache-Group), beeinflussen in diesem Fall allerdings das Füllverhalten nicht.

Abschließend wollen wir uns noch mit Zusammenhängen zwischen verschiedenen Teilmengen beschäftigen. Einen solchen Zusammenhang haben wir bereits kennen gelernt. Hier zerlegten umgebungsvollständige und umgebungsfragmentarische Werte zusammen mit ungeladenen Werten die DBD einer Spalte. Die Menge $V_{l|c|\tau}$ der zum Zeitpunkt τ in c geladenen Werte kann ebenfalls partitioniert werden:

Folgerung 1: Es gilt:

$$V_{l|c} = V_{vc|c|\tau} \cup V_{vf|c|\tau} = V_{s|c|\tau} \cup V_{i|c|\tau} \quad \text{für alle } \tau \in F, c \in \mathbf{COL} \qquad (2.4.4)$$

Dabei bilden die beiden Mengen der umgebungsvollständigen bzw. umgebungsfragmentarischen Werte eine Partitionierung der Menge der geladenen Werte.

Beweis: Sowohl für umgebungsfragmentarische als auch für umgebungsvollständige Werte gilt nach Definition, dass mindestens ein Satz, der einen solchen Wert in der betrachteten Spalte hat geladen ist. Damit ist ein solcher Wert auch in der Menge der geladenen Werte. Umgekehrt ist für einen Wert die Voraussetzung, um geladen zu sein, dass mindestens ein Satz aus seiner Werteumgebung in der Cache-Tabelle geladen ist. Ein geladener Wert muss also zumindest umgebungsfragmentarisch sein, kann aber auch umgebungsvollständig sein. Die beiden Mengen der umgebungsfragmentarischen bzw. umgebungsvollständigen Werte sind disjunkt. Somit bilden sie eine Partitionierung der Menge der geladenen Werte.

Ein geladener Wert muss aber entweder induziert oder geschmuggelt sein und jeder induzierte oder geschmuggelte Wert ist per Definition geladen. ■

Die Mengen der geschmuggelten und induzierten Werte sind (im Gegensatz zu den Mengen der umgebungsvollständigen und umgebungsfragmentarischen Werte) nicht unbedingt disjunkt. Ein Wert kann durchaus in einem Füllschritt geschmuggelt und in einem Füllschritt in die gleiche Spalte induziert worden sein.

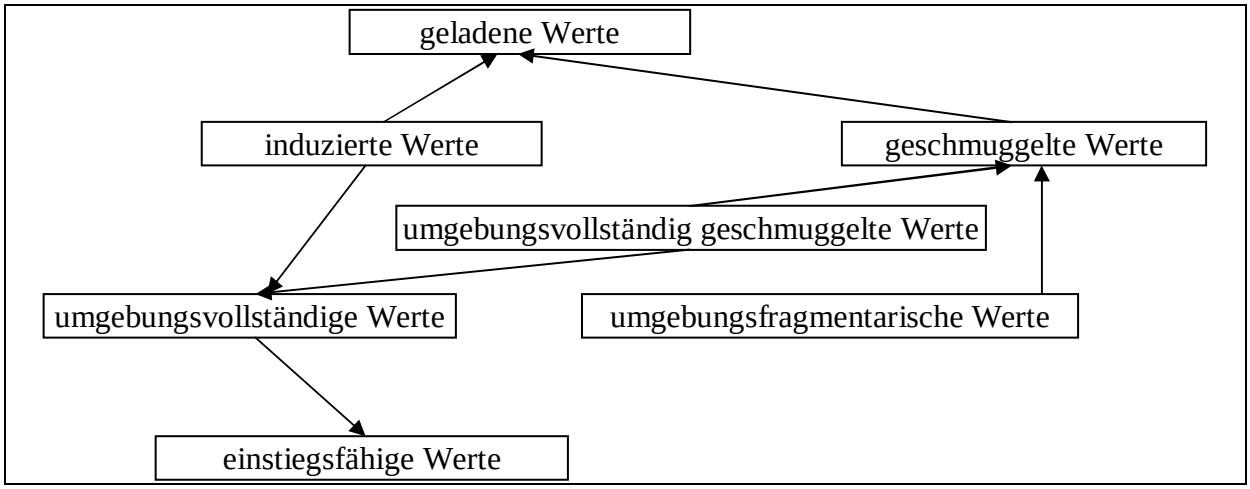

Abbildung 6: Beziehungen der verschiedenen Eigenschaften von Werten untereinander.

Abbildung 6 zeigt die verschiedenen Beziehungen von Werten untereinander. Die Pfeile symbolisieren dabei jeweils eine Beziehung „sind ebenfalls“, d. h. die Werte am Ende eines Pfeils liegen ebenfalls in der Menge am Anfang des Pfeils (d. h. sie haben die Eigenschaft). Man kann die Pfeile also auch als Teilmengenbeziehungen deuten. Folgende Beziehungen können identifiziert werden:

- induzierte und geschmuggelte Werte sind insbesondere auch geladen,
- induzierte Werte sind außerdem umgebungsvollständig und diese wiederum sind einstiegsfähig,
- es kann eine Schnittmenge zwischen geschmuggelten und umgebungsvollständigen Werten geben (dargestellt durch die Menge der umgebungsvollständig geschmuggelten Werte als Teilmenge der geschmuggelten und umgebungsvollständigen Werte),
- umgebungsfragmentarische Werte sind geschmuggelt.

2.5 Exkurs: Abschätzung der Schnittmengenmächtigkeit

Bei der Modellierung von Cache-Groups und -Föderationen ist oft die Mächtigkeit einer Schnittmenge zu berechnen. Dies ist ein Grund, warum wir uns in diesem Exkurs mit der Abschätzung der Schnittmengenmächtigkeit beschäftigen.

Teilmengen-Beziehungen bieten uns einen Ansatzpunkt, um die Mächtigkeit einer Schnittmenge zu berechnen. Ist eine Menge A Teilmenge einer zweiten Menge B so gilt offensichtlich:

$$|A \cap B| = |A|.$$

In unseren Modellen gibt eine große Anzahl von Teilmengen-Beziehungen zwischen den verschiedensten Mengen. Zu nennen sind hier noch einmal die in Abschnitt 2.4 identifizierten Teilmengen-Beziehungen, aber für Satzmengen können wir ebenfalls Teilmengen-Beziehungen identifizieren: So handelt es sich bei einer Werteumgebung immer um eine Teilmenge der im Backend für eine Relation gespeicherten Sätze. Ebenso ist die Menge der in eine Cache-Tabelle geladenen Sätze eine Teilmenge dieser Menge.

Aber auch wenn A keine Teilmenge von B ist, können wir unter einer Voraussetzung die Schnittmengenmächtigkeit abschätzen[12]: Beide Grundmengen der Schnittmenge müssen dabei jeweils eine Teilmenge einer gemeinsamen Obermenge C sein.

Wir geben davon aus, dass die Ereignisse, dass ein gegebenes Element aus C in A bzw. in B liegt unabhängig sind. Natürlich gilt diese Annahme je nach Wahl von C im Allgemeinen nicht. Da wir aber davon ausgehen, dass wir keine Informationen über die konkrete Zusammensetzung der Mengen haben, ist diese Annahme der einzige Ansatzpunkt. Der Fehler der Abschätzung hängt also maßgeblich von der Wahl der gemeinsamen Obermenge ab.

Natürlich fällt die Abschätzung der Schnittmengenmächtigkeit genauer aus, wenn wir eine kleinere Obermenge nehmen. Haben wir also die Wahl zwischen zwei verschiedenen Obermengen C und D, die sich gegenseitig einschließen und beide die Grundmengen A und B beinhalten, so ist es ratsam, die kleinere der beiden Teilmengen zu wählen. Es gilt nämlich:

$$A \cup B \subseteq C \subseteq D.$$

Die Verwendung von C beinhaltet implizit die zusätzliche Information, dass beide Mengen A und B Teilmengen sowohl von C und D sind. Dieses Mehr an Information erhöht natürlich die Genauigkeit der Abschätzung der Schnittmengenmächtigkeit. Formal ist die Wahrscheinlichkeit für ein Element aus D zu $A \cap B$ zu gehören gleich $|A \cap B|/|D|$. Dies gilt natürlich insbesondere für die Elemente aus C, die ebenfalls D gehören. Für die Elemente aus der Differenzmenge zwischen C und D ist die Wahrscheinlichkeit, zu $A \cap B$ zu gehören gleich null, denn die sowohl die Elemente aus A als auch die Elemente aus B liegen in D. Mit dem zusätzlich gewonnenen Wissen können wir (bzw. müssen wir) von der Gleichverteilungsan-

[12] Mathematisch korrekt wäre hier der Begriff annähern, da eine Abschätzung obere und untere Schranken liefert. Wir nähern den Wert allerdings im Sinne einer Average-Case-Analyse an.

nahme abrücken und können so die Wahrscheinlichkeit nicht mehr über mit C als gemeinsamer Obermenge berechnen.

Nehmen wir also im Folgenden an, wir haben die optimale Wahl für C getroffen, so dass die Annahme der Unabhängigkeit der beiden Mengen im der gemeinsamen Obermenge gilt. Die Wahrscheinlichkeit für ein Element aus C, in A zu liegen, sei für jedes Element gleich groß. Gleiches gelte für die Wahrscheinlichkeit, in B zu liegen: Auch diese sei gleich groß für jedes Element aus C.

Wir betrachten zunächst die Wahrscheinlichkeit, dass ein gegebenes Element x aus C in der Menge A liegt. Analog verfahren wir für die Wahrscheinlichkeit, dass x in B liegt. Um die Verteilungsfunktion für diese Wahrscheinlichkeit zu berechnen, teilen wir die Anzahl der günstigen Fälle durch die Anzahl der möglichen Fälle:

$$p(x \in A) = |A| / |C| \qquad p(x \in B) = |B| / |C|$$

Die beiden Ereignisse sind unabhängig, denn ob ein gegebenes Element in Menge A liegt, hat nach Annahme keinen Einfluss darauf, ob dieses Element auch in B liegt. Somit können wir die Wahrscheinlichkeiten multiplizieren, um die Wahrscheinlichkeit für das Ereignis zu erhalten, dass x in der Schnittmenge aus A und B liegt. Formal gilt:

$$p(x \in A \cap B) = p(x \in A \wedge x \in B) = p(x \in A) \cdot p(x \in B) = |A| / |C| \cdot |B| / |C| = |A| \cdot |B| / |C|^2.$$

Nun definieren wir eine Zufallsvariable I_x, die angibt, ob ein gegebenes Element x in der Schnittmenge zwischen A und B liegt. Wir schreiben:

$$I_x = \begin{cases} 1 \text{ falls } x \in A \cap B \\ 0 \text{ sonst} \end{cases}.$$

Wir können nun den Erwartungswert von I_x berechnen:

$$E(I_x) = 1 \cdot p(x \in A \cap B) + 0 \cdot p(x \notin A \cap B) = p(x \in A \cap B) = |A| \cdot |B| / |C|^2.$$

Um nun die Mächtigkeit der Schnittmenge zwischen A und B zu erhalten, definieren wir eine Zufallsvariable, die die Anzahl der Elemente zählt, die in der Schnittmenge liegen. Wir schreiben:

$$Z = \sum_{x \in C} I_x .$$

Eine Abschätzung der Schnittmengenmächtigkeit liefert uns der Erwartungswert der Zufallsvariable Z. Es gilt:

$$|A \cap B| \approx E(Z) = E(\sum_{x \in C} I_x) = \sum_{x \in C} E(I_x) = \sum_{x \in C} |A| \cdot |B| / |C|^2 .$$

Da alle Summanden unabhängig von der konkreten Wahl von x sind können wir die Summe vereinfachen:

$$|A \cap B| \approx \sum_{x \in C} |A| \cdot |B| / |C|^2 = |C| \cdot (|A| \cdot |B| / |C|^2) = |A| \cdot |B| / |C| \qquad (2.5.1)$$

Wir können die Formel auch durch den Erwartungswert der hypergeometrischen Verteilung begründen.

Definition: Zu einer Menge C gebe es zwei voneinander unabhängige Teilmengen A und B. Die **hypergeometrische Wahrscheinlichkeitsverteilung** mit den Parametern |A|, |B| und |C|, $\Phi(|A|, |B|, |C|)$, gibt die Wahrscheinlichkeit dafür an, dass die beiden Teilmengen genau 0, 1, 2, 3, ... Elemente gemeinsam besitzen [Wiki04a].

Der Erwartungswert der Zufallsvariable I mit hypergeometrischer Verteilung $\Phi(|A|, |B|, |C|)$ lässt sich berechnen durch:

$$E(I) = |A| \cdot |B| / |C|.$$

Genauigkeit der Abschätzung

Um zu sehen, wie gut die Näherung ist, suchen wir nun nach einer Schranke für den Fehler. Als obere und untere Grenze der Annäherung können wir die beiden Extremfälle betrachten.

Die Schnittmenge mit der größten Mächtigkeit ergibt sich sicherlich, wenn die beiden Mengen in einer Teilmenge-Beziehung stehen:

$$|A \cap B| \leq \min(|A|, |B|).$$

Für die untere Schranke müssen wir zwei Fälle unterscheiden.

- Zum einen kann es sein, dass die Mächtigkeit der gemeinsamen Obermenge C kleiner als die Summe der Mächtigkeiten von A und B ist. In diesem Fall können A und B nicht disjunkt sein. Unter dieser Voraussetzung erhalten wir die untere Schranke, wenn wir annehmen, dass C gleich der Vereinigung von A und B ist. Das Prinzip von Inklusion und Exklusion[13] liefert uns in diesem Fall:

$$|C| < |A| + |B| \Rightarrow |A \cap B| > 0 \Rightarrow |A \cap B| \geq |A| + |B| - |C|.$$

- Der andere Fall ergibt sich, wenn |C| größer oder gleich Summe der Mächtigkeiten von A und ist. In diesem Fall können A und B disjunkt sein, was die kleinste mögliche Schnittmenge und damit die untere Schranke darstellt:

$$|C| \geq |A| + |B| \Rightarrow |A \cap B| \geq 0.$$

Eine Fehlerabschätzung können wir dementsprechend finden, wenn wir den Maximalwert der Differenz zur oberen bzw. unteren Schranke betrachten. Es gilt:

$$\text{abs}(|A \cap B| - |A| \cdot |B| / |C|) \leq \max(\min(|A|, |B|) - |A| \cdot |B| / |C|, |A| \cdot |B| / |C| - \max(|A| - |B| + |C|, 0).$$

[13] Wenn wir die Mächtigkeit der Vereinigung von A und B berechnen wollen, so kann dies geschehen, indem wir die Elemente aus A bzw. B zählen und die Summe der Mächtigkeiten bilden. Dann haben wir aber einige Elemente doppelt gezählt, nämlich genau die, die in der Schnittmenge zwischen A und B liegen. Diese Elemente müssen wir dementsprechend wieder abziehen.

Mit Hilfe der vorgeschlagenen Methode um die Schnittmengenmächtigkeit abzuschätzen, können wir ebenfalls die Mächtigkeit der Vereinigung zweier Mengen abschätzen, solange die beiden Mengen Teilmengen einer gemeinsamen Obermenge sind

$$|A \cup B| = |A| + |B| - |A \cap B| \approx |A| + |B| - |A| \cdot |B| / |C|.$$

2.6 Histogramme und Zugehörigkeitsfunktion

Im Rahmen dieser Arbeit werden Histogramme zur Verbesserung der Abschätzung des Füllverhaltens bei baumartigen Cache-Groups, also in Abschnitt 3, eingesetzt. Bei allgemeinen, insbesondere zyklenbehafteten Cache-Groups würde die Modellierung hierdurch so komplex, dass die Verbesserung der Abschätzung die Komplexität der Modellierung nicht rechtfertigen würde. In den Betrachtungen über baumartige Cache-Groups werden wir dementsprechend intensiv den Einfluss von Histogrammen auf mögliche Abschätzungen betrachten. In weiteren Modellen für allgemeine Cache-Groups und Cache-Föderationen werden wir nur am Rande auf Histogramme eingehen. An Stellen, an denen Histogramme prinzipiell einen verbessernden Einfluss auf die Abschätzung des Füllverhaltens ausüben können, erwähnen wir dies dennoch kurz.

Definition: Ein Histogramm h_c von Spalte c ist eine Abbildung mit folgender Gestalt [Ioa03]:

$$h_c : V \rightarrow \mathbb{N},\ h_c(\varphi) = |S_{c,\varphi}|. \qquad (2.6.1)$$

Das Histogramm h_c bildet Werte φ auf die Mächtigkeit ihrer Werteumgebung $S_{c,\varphi}$ ab.[14] Anschaulich liefert das Histogramm eines Werts also seine Häufigkeit in der Spalte. Diese Häufigkeitsverteilung wird oft graphisch durch Balkendiagramme dargestellt. Tatsächlich zeugt der Name „Histogramm“ von dieser Bedeutung: einer graphischen Darstellung der Häufigkeit von (Mess-)Werten.

Der Definitionsbereich jedes Histogramms ist die Menge V aller möglichen Werte. Für Werte außerhalb der DBD der Spalte c liefert das Histogramm natürlich die Häufigkeit null, da der Wert null-mal in der Spalte vorkommt.

In der Praxis haben sich verschiedene Arten von Histogrammen durchgesetzt. Reale Histogramme liefern jeweils nur einen Schätzwert der Häufigkeit eines betrachteten Werts. Ein guter Überblick über die verschiedenen Arten von Histogrammen ist in [Ioa03] zu finden. Es sei noch angemerkt, dass zu jeder Art von Histogramm eine Möglichkeit gefunden werden kann, um die hier verwendete Form zu erhalten. D. h. man kann zu einem gegebenen Histogramm einer Spalte c und einem Wert φ algorithmisch den in dieser Arbeit verwenden Wert $h_c(\varphi)$ erhalten.

Man beachte, dass unter der Gleichverteilungsannahme die Mächtigkeiten der Werteumgebungen aller Werte gleich groß ist. Wir können in diesem Fall die Histogrammwerte durch

[14] Für eine Erklärung der Schreibweisen sei auf Anhang B (wichtige Schreibweisen und Symbolverzeichnis) verwiesen.

den Quotienten aus der Cache-Tabelle-Kardinalität $|S_{T(c)}|$ und der Spalten-Kardinalität c_c (also der Mächtigkeit der DBD der Spalte) berechnen:

$$h_c(\varphi) = |S_{c,\varphi}| = |S_{T(c)}| / c_c \text{ für alle } \varphi \in V_c. \qquad (2.6.2)$$

Mit Histogrammen können wir ebenfalls die Mächtigkeit der Werteumgebung von Wertemengen berechnen, wenn diese aus umgebungsvollständigen Werten besteht:

$$|S_{vc|c}| = \sum_{\varphi \in V_{vc|c}} h_c(\varphi) . \qquad (2.6.3)$$

Analog ist diese Berechnungsvorschrift auch für Teilmengen dieser zugrunde liegenden Wertemenge möglich. So kann z. B. für die Menge der induzierten Werte die Mächtigkeit ihrer Werteumgebung wie folgt berechnet werden:

$$|S_{i|c}| = \sum_{\varphi \in V_{i|c}} h_c(\varphi) . \qquad (2.6.4)$$

Unter der Gleichverteilungsannahme können wir die Summe einfacher berechnen, da alle Summanden gleich groß sind. Als Beispiel nehmen wir wieder die Menge der umgebungsvollständigen Werte in c:

$$|S_{vc|c}| = |V_{vc|c}| \cdot |S_{T(c)}| / c_c. \qquad (2.6.5)$$

Ebenfalls mit Hilfe der Histogramme können wir eine weitere nützliche Funktion definieren.

Definitionen:

- Sei c eine Spalte, φ ein Wert. Die **Zugehörigkeitsfunktion** $\chi_c(\varphi)$ gibt an, ob φ in der DBD von c vorhanden ist:

$$\chi_c(\varphi) = \begin{cases} 1 \text{ wenn } \varphi \in V_c \\ 0 \text{ sonst} \end{cases} = \max(1, h_c(\varphi)). \qquad (2.6.6)$$

- Außerdem definieren wir für eine Menge von Spalten $\{c_1,\ldots,c_n\}$ die **kombinierte Zugehörigkeitsfunktion** $\chi_{c_1 \cdot \ldots \cdot c_n}$. Die kombinierte Zugehörigkeitsfunktion gibt an, ob der Wert in den DBDs aller Spalten vorhanden ist:

$$\chi_{c_1 \cdot \ldots \cdot c_n}(\varphi) = \chi_{c_1}(\varphi) \cdot \ldots \cdot \chi_{c_n}(\varphi) = \begin{cases} 1 \text{ wenn } \varphi \in V_{c_1} \cap \ldots \cap V_{c_n} \\ 0 \text{ sonst} \end{cases} . \qquad (2.6.7)$$

2.7 Verhalten von Cache-Keys

Cache-Keys sind vom Benutzer (bzw. Datenbank-Administrator) deklarativ definierte Spalten mit der Einstiegspunkteigenschaft und der Füllpunkteigenschaft. Auf Grund der Füllpunkteigenschaft wird bei einem Cache-Miss ein Füllzyklus ausgelöst. In einem ersten Füllschritt wird dabei die Werteumgebung des referenzierten Cache-Key-Werts in die Cache-Tabelle des Cache-Keys geladen. Cache-Keys haben also einen direkten Einfluss auf das Füllverhalten einer Cache-Group.

Im Folgenden wollen wir das Verhalten eines Cache-Keys bei einem Füllschritt betrachten. Wir schauen uns also an, wie viele Sätze in die Wurzel-Tabelle der Cache-Group durch das Referenzieren eines Cache-Key-Werts φ_{ck} geladen werden.

Die Mächtigkeit der Werteumgebung hängt natürlich vom durch die Benutzeranfrage referenzierten Cache-Key-Wert ab. Da wir das Füllverhalten der Wurzel-Tabelle allgemein modellieren wollen, schätzen wir einen Mittelwert $s_{ref|ck}$ für die Mächtigkeit der Werteumgebung (d. h. die Häufigkeit) des Cache-Key-Werts ab.

Um diesen Mittelwert abzuschätzen, existieren zwei Alternativen. Zum einen ist es möglich, davon auszugeben, dass jeder Cache-Key-Wert gleich oft durch Benutzeranfragen referenziert wird und zum Laden von Sätzen führt. Zum anderen verwenden wir eine vorgegebene Referenzierungswahrscheinlichkeit mit einer Verteilungsfunktion.

Betrachten wir den Fall der gleichmäßig häufigen Referenzierung aller Cache-Key-Werte. In dieser Situation ist die Berechnung $s_{ref|ck}$ sehr einfach, denn dann ist diese Größe unabhängig von der konkreten Wahl des Cache-Key-Werts und hängt nur von der Kardinalität der Spalte und der Kardinalität der Wurzel-Tabelle ab. Betrachten wir den Füllschritt der Wurzel-Tabelle im Detail. Dann können wir feststellen, dass Histogramme keinen Einfluss auf den abgeschätzten Ladestand der Wurzel-Tabelle haben. Formal gilt:

Folgerung 2: Sei ck ein Cache-Key. Die Referenzierungswahrscheinlichkeit der Cache-Key-Werte sei gleichverteilt. Dann gilt:

$$s_{ref|ck} = |S_{T(ck)}| / c_{ck}. \qquad (2.7.1)$$

Beweis: Nehmen wir an, wir wissen, dass der Wert φ_{ck} referenziert wurde. Unter der Verwendung eines Histogramms können wir die Mächtigkeit der Werteumgebung von φ_{ck} direkt bestimmen. Es gilt:

$$s_{ref|ck} = h_{ck}(\varphi_{ck}). \qquad (2.7.2)$$

Da wir das Füllverhalten allgemein, also unabhängig von der konkreten Wahl eines Cache-Key-Werts, modellieren wollen, müssen wir eine durchschnittliche Anzahl an in die Wurzel-Tabelle geladenen Sätzen berechnen. Wir bilden also den Erwartungswert indem wir über die Häufigkeit $h_{ck}(\varphi)$ jedes möglichen Cache-Key-Wert φ summieren und teilen diese Summe durch die Anzahl der betrachteten Werte, also die Kardinalität c_{ck} des Cache-Keys:

$$s_{ref|ck} = \sum_{\varphi \in V_{ck}} h_{ck}(\varphi) \; / \; c_{ck} = |S_{T(ck)}| / c_{ck}. \; \blacksquare \qquad (2.7.3)$$

Diese Berechnungsvorschrift ist unabhängig von der Verteilung der Werte im Cache-Key. Man beachte allerdings, dass der beschriebene Zusammenhang nur gilt, wenn wir annehmen, dass alle Cache-Key-Werte gleich oft referenziert werden.

Können wir nicht von der Annahme ausgehen, dass die Referenzierungswahrscheinlichkeit der Cache-Key-Werte gleichverteilt ist, da z. B. bestimmte Cache-Key-Werte überproportional oft referenziert werden, kann die durchschnittliche Werteumgebung des referenzierten Cache-Key-Werts nur mit Hilfe der Verteilungsfunktion $p_{ref|ck}$ und des Histogramms h_{ck} abgeschätzt werden:

$$s_{ref|ck} = \sum_{\varphi \in V_{ck}} (h_{ck}(\varphi) \cdot p_{ref|ck}(\varphi)) . \qquad (2.7.4)$$

Diese Formel kann als Spezialfall von Formel 2.7.1 betrachtet werden. Denn ist die Referenzierungswahrscheinlichkeit für die Cache-Key-Werte gleichverteilt, so gilt für die Verteilungsfunktion:

$$p_{ref}(\varphi) = 1 / c_{ck} \text{ für alle } \varphi \in V_{ck}.$$

Durch Einsetzen erhält man die Berechnungsvorschrift aus Folgerung 2.

2.8 RCCs

Wie wir gesehen haben, können über zwei Wege Sätze in eine Cache-Tabelle gelangen: Zum einen kann durch einen Cache-Miss das Laden der Werteumgebung des referenzierten Cache-Key-Werts angestoßen werden, zum anderen kann es notwendig sein, dass Sätze in eine Cache-Tabelle am Ende einer RCC geladen werden, um die RCC zu erfüllen. RCCs haben also einen großen Einfluss auf das Füllverhalten einer Cache-Group oder -Föderation. Dementsprechend wollen wir in diesem Abschnitt RCCs und ihren Einfluss auf das Füllverhalten genauer betrachten.

RCCs sind vom Datenbank-Administrator deklarativ eingerichtete Objekte, die über ihre Eigenschaft das Zwischenspeichern von Extensionen von Verbund-Anfragen ermöglichen. Durch die RCC-Eigenschaft wird gewährleistet, dass Verbundanfragen aus dem Cache beantwortet werden können. Dafür muss jeder Wert, der in einer Spalte geladen ist, im Nachfolger dieser Spalte einstiegsfähig sein.

2.8.1 Die RCC-Eigenschaft

Damit Verbundanfragen aus dem Cache beantwortet werden können, muss jeder Wert, der in einer Spalte geladen ist, in ihrem Nachfolger einstiegsfähig sein. Der Cache-Manager setzt diese Eigenschaft als Constraint durch und lädt gegebenenfalls Sätze aus dem Backend, um die Constraint zu erfüllen (und den Cache somit in einen stabilen, gültigen Zustand zu bringen).

Formal können wir die RCC-Eigenschaft nun folgendermaßen beschreiben:

Definition: Seien e und k zwei Spalten mit e→k. Die **RCC-Eigenschaft** besagt, dass nach dem Füllzyklus alle in e geladenen Werte in k einstiegsfähig sein müssen:

$$\text{Entweder } \varphi \in V_{i|k} \quad \text{oder} \quad \varphi \notin V_k \quad \text{für alle } \varphi \in V_{l|e}. \qquad (2.8.1.1)$$

Man beachte, dass ein induzierter Wert nach Definition insbesondere umgebungsvollständig und damit natürlich auch einstiegsfähig ist. Werte, die zwar im Vorgänger geladen aber nicht in der DBD des Konsumenten vorhanden sind, werden natürlich nicht induziert und können nie umgebungsvollständig sein (trotzdem sind diese einstiegsfähig).

Die nächste Folgerung besagt, dass generell alle ankommenden Werte induziert werden, solange sie in der DBD der aktiven Spalte vorkommen, aber in dieser nicht umgebungsvollständig sind.

Folgerung 3: Sei c eine Spalte. Betrachten wir einen RCC-Füllschritt für die RCC e→c, dann gilt:

$$\varnothing \neq V'_{i|c|\tau} \subseteq (V_c \cap V_{l|e|\tau}) - V_{vc|c|\tau}. \qquad (2.8.1.2)$$

Begründung: Bei RCC-Füllschritten wird mindestens ein Wert induziert, denn ansonsten würde der Füllschritt den Zustand der Cache-Tabelle nicht verändern und wir könnten ihn vernachlässigen.[15] Für die zum Zeitpunkt τ zusätzlich induzierten Werte gilt, dass sie in der DBD der aktiven Spalte vorhanden, aber vor τ nicht umgebungsvollständig waren. Außerdem sind diese Werte im Vorgänger der aktiven Spalte geladen. Allerdings müssen bei direkten RCC-Füllschritten nicht alle diese Werte induziert werden, sondern nur eine Teilmenge davon[16]. ■

Durch die RCC-Eigenschaft wird also garantiert, dass die Extension eines Prädikats mit einem Verbund „e = k“ im Cache ist, wenn e→k gilt und die Extension des Prädikats, das durch Weglassen von (mindestens diesem) Verbund entsteht, im Cache ist.

Die nächste Folgerung besagt, dass sich die Menge $V_{i|c|\tau'}$ der insgesamt in eine Spalte induzierten Werte iterativ aus den Mengen $V'_{i|c|\tau}$ der bei den Füllschritten jeweils zusätzlich induzierten Werte bestimmen lässt.

Folgerung 4: Sei c eine Spalte und τ und τ' zwei aufeinander folgende Zeitpunkte, wobei T(c) zum Zeitpunkt τ einen Füllschritt ausführe. Dann gilt:

$$V_{i|c|\tau'} = V_{i|c|\tau} \cup V'_{i|c|\tau} \quad \text{und} \quad V_{i|c|\tau} \cap V'_{i|c|\tau} = \varnothing \qquad (2.8.1.3)$$

Begründung: Nach dem betrachteten Füllschritt, also zum Zeitpunkt τ' sind alle Werte in c induziert, die vorher bereits induziert waren, zuzüglich denen, die während des Füllschritts zusätzlich induziert wurden. Dass die Menge der während des Füllschritts zusätzlich induzierten Werte disjunkt zur Menge der vorher bereits induziert Werte ist, können wir uns anhand der vorher gezeigten Formel bewusst machen. Dies ergibt sich aber auch aus der Definition der zusätzlich geladenen Werte (ein induzierter Wert ist insbesondere geladen). ■

Mit Hilfe der Zugehörigkeitsfunktion können wir die RCC-Eigenschaft für einzelne Werte auch wie folgt erklären: Seien e und k zwei Spalten mit e→k. Dann besagt die RCC-Eigenschaft:

---

[15] Von einem tatsächlichen Datenbanken-Caching-System mit Cache-Groups wird eventuell dennoch ein Füllschritt ausgeführt, da im Allgemeinen nicht geprüft wird, ob es dadurch zu einer Veränderung kommt. Wir wissen allerdings, dass sich durch einen solchen Füllschritt der Füllstand der Cache-Tabelle nicht verändert und somit kann der Füllschritt durch Algorithmen zur Berechnung der Füllstände ignoriert werden.

[16] Bei zusammenfassenden RCC-Füllschritten gilt dies nicht, sondern es werden immer *alle* ankommenden Werte während des Füllschritts einstiegsfähig gemacht. Dementsprechend können wir das Teilmengen-Symbol im bei zusammenfassenden RCC-Füllschritten durch ein Gleichheitszeichen ersetzen.

$$\varphi \in V_{l|e} \wedge \chi_k(\varphi) = 1 \Rightarrow \varphi \in V_{i|k}.$$

Die Formel besagt also, dass ein Wert φ in einen Konsumenten k nach dem Füllzyklus induziert ist, wenn er in seinem Vorgänger e geladen ist und in der DBD von k liegt.[17]

Den dargestellten Zusammenhang nennen wir **RCC-Implikation**. Im Allgemeinen gilt die Implikation tatsächlich nur in die gezeigte Richtung. Nur in Sonderfällen kann die RCC-Implikation auch für die andere Richtung gezeigt werden. Dies gilt unter anderem für isolierte RCCs.

Definition: Wir nennen eine RCC e→k **isoliert**, wenn k kein Cache-Key ist, keine weitere RCC und keine Schmuggler-Beziehung bei k endet.

Isolierte RCCs haben die angenehme Eigenschaft, dass alle in den Konsumenten am Ende der RCC induzierten Werte über die isolierte RCC induziert werden. Wenn wir eine Teilmenge der Menge von induzierten Werten betrachten, gilt für diese die RCC-Implikation ebenfalls in beide Richtungen. Eine Cache-Tabelle mit einer eingehenden isolierten RCC hat weiterhin lediglich einen Konsumenten. Gäbe es neben dem Konsumenten mit der eingehenden isolierten RCC einen zweiten Konsumenten, so wäre zwischen diesen beiden Konsumenten eine gegenseitige Schmuggler-Beziehung und die RCC wäre nicht isoliert.

Definition: Seien e und k zwei Spalten mit e→k. Wir nennen die Menge $V_{i|e\to k}$ der in e geladenen und k induzierten Werte **induziert über e→k**.

Folgerung 5: Sei e→k eine RCC. Dann gilt für alle Zeitpunkte τ:

$$V_{i|k} \supseteq V_{i|e\to k}, \quad V'_{i|k|\tau} = V'_{i|e\to k|\tau} \text{ und } \quad \varphi \in V_{i|e\to k} \Leftrightarrow \varphi \in V_{l|e} \wedge \chi_k(\varphi) = 1 \qquad (2.8.1.1)$$

Beweis: Allgemein werden während eines Füllschritts alle Werte induziert, die in einem beliebigen Vorgänger geladen sind, also auch insbesondere die, die in e geladen sind. Somit gilt auch $V_{i|e\to k} \subseteq V_{i|k}$. Jeder über die RCC e→k induzierte Wert muss aber insbesondere in e geladen sein. Somit gilt die RCC-Implikation in beide Richtungen. ■

Die RCC-Eigenschaft kann ebenfalls im laufenden Betrieb des Caches dazu verwendet werden, um zu prüfen, ob ein Wert in einem Konsumenten umgebungsvollständig ist. Wir erinnern uns, dass alle induzierten Werte umgebungsvollständig und damit insbesondere einstiegsfähig sind. Wir können einen Konsumenten also für einen solchen Wert als Einstiegspunkt benutzen. Die nächste Folgerung zeigt uns, wie wir die RCC-Eigenschaft für dieses **Sondierung** genannte Vorhaben verwenden können.

Folgerung 6: Ist k nicht der Cache-Key, so sind alle induzierten Werte zu dem betrachteten Zeitpunkt τ in einem Vorgänger von k geladen.

Beweis: Nehmen wir das Gegenteil an, dann existiert mindestens ein Wert, der in die Spalte induziert ist, der aber in keinem Vorgänger geladen wurde. Dieser Wert müsste während eines

[17] Man beachte, dass diese Bedingung lediglich hinreichend, nicht aber notwendig ist: ein Wert kann auch dann induziert sein, wenn der nicht in *diesem* Vorgänger geladen ist (er kann auch in einem *anderen* Vorgänger geladen sein). Ein Teil der Bedingung ist allerdings notwendig, nämlich, dass φ in der DBD des Konsumenten k liegt.

Füllschritts in c geladen worden sein. Da der Wert geladen wurde, muss er bei der Spalte angekommen sein. Hierfür muss er aber in einem Vorgänger geladen gewesen sein und in c ungeladen. Dies widerspricht aber der Voraussetzung. ■

Die Sondierung erfolgt also daraufhin, ob der Parameter des Gleichheitsprädikats in der Benutzeranfrage in einem Vorgänger der Spalte geladen ist. Bei einem positiven Ergebnis der Sondierung können wir den Konsumenten als Einstiegspunkt benutzen, da die Extension des Anfrageprädikats komplett im Cache ist. Durch diese Art der Sondierung kann man wesentlich mehr Anfragen aus dem Cache beantworten als bei der „klassischen" Vorgehensweise. Bei der ursprünglichen Art der Sondierung konnte man nur komplett einstiegsfähige Spalten als Einstiegspunkte benutzen. Hier kann für bestimmte Werte auch eine Spalte am Ende eines RCCs als Einstiegspunkt benutzt werden.

Bei Cache-Keys könnte die Sondierung so funktionieren, dass eine zusätzliche Kontrolltabelle T_{ckc} die im Cache-Key induzierten Cache-Key-Werte in einer Spalte ckc, die unique ist und über eine RCC ckc→ck mit dem Cache-Key ck verbunden ist, speichert. Die Sondierung könnte dann über den gleichen Mechanismus wie bei Konsumenten mit Hilfe der Kontrollspalte geschehen. Die RCC-Eigenschaft wäre dann implizit für diese RCC ebenfalls erfüllt.

Bei negativem Ausgang muss wegen der Füllpunkt-Eigenschaft des Cache-Keys ein neuer Füllzyklus ausgeführt werden, um die Extension in den Cache zu holen. Nach dem Füllzyklus würde der Parameter des Gleichheitsprädikats in die Kontrollspalte eingetragen, um zu signalisieren, dass Benutzeranfragen mit diesem Parameter zukünftig aus dem Cache beantwortet werden können. Dies gilt ebenfalls dann, wenn durch den Füllzyklus keine neuen Sätze in den Cache geladen werden, weil der Cache-Key-Wert nicht in der DBD des Cache-Keys vorhanden ist.

Auf diese Art kann die Sondierung sogar noch effektiver stattfinden als bei der klassischen Variante. Die Vorteile sind:

- Es können wesentlich mehr Spalten als Einstiegspunkt verwendet werden. Prinzipiell stehen hierfür alle Konsumenten zur Verfügung.
- Durch die Sondierung in der Kontrollspalte ist es sogar möglich, ein so genanntes negatives Caching durchzuführen [Büh04]. Hierbei kann durch den Cache erkannt werden, wenn die Extension des Anfrageprädikats eine leere Satzmenge ist und diese als Ergebnis an den Benutzer geschickt werden. Beim klassischen Caching würde ein Cache-Miss vermutet, da der Cache-Key-Wert nicht zwischengespeichert wurde und die Anfrage würde vom Cache-Manager zur Backend-DB weitergeleitet.

2.8.2 Die RCC-Teilmengen-Annahme

Wie wir in Abschnitt 2.8.1 gesehen haben, sind einige Werte, die in einem Erzeuger geladen sind, nach dem Füllzyklus im entsprechenden Konsumenten nicht umgebungsvollständig. Die Ursache hierfür ist, dass der Wert nicht im Backend des Konsumenten vorhanden ist, d. h. seine Werteumgebung ist leer.

Ein Wert, der nach dem Füllzyklus in einem Erzeuger geladen und im entsprechenden Konsumenten umgebungsvollständig ist, muss in der Schnittmenge zwischen den DBDs der beiden Spalten vorhanden sein.

Um dass Füllverhalten einer Cache-Group oder -Föderation zu modellieren, müssen wir oft eine Aussage darüber treffen, ob ein ankommender Wert in der DBD des Konsumenten liegt. Haben wir keine Informationen über die Mächtigkeit der Schnittmenge der DBDs von Erzeuger und Konsumenten, so müssen wir eine Annahme hierzu treffen. Nur so kann eine Wahrscheinlichkeit dafür angegeben werden, ob ein Wert zu einem Laden von Sätzen in die Cache-Tabelle führt.

Definition: Seien e und k zwei Spalten mit e→k. Die **RCC-Teilmengen-Annahme** besagt, dass dann die kleinere der beiden DBDs eine Teilmenge der größeren DBD ist. Formal gilt:

$$c_e \leq c_k \Leftrightarrow V_e \subseteq V_k \quad \text{und} \quad c_e \geq c_k \Leftrightarrow V_e \supseteq V_k. \qquad (2.8.2.1)$$

In den meisten realen Fällen dürfte die RCC-Teilmengen-Annahme dem Sachverhalt bei den zu modellierenden Cache-Groups entsprechen. Unter üblichen Umständen werden RCCs nämlich für Fremdschlüsselbedingungen zwischen den beteiligten Spalten definiert, auch wenn dies prinzipiell nicht notwendig ist.

Existiert eine Fremdschlüsselbedingung vom Fremdschlüssel einer Tabelle zum Primärschlüssel einer anderen Tabelle, so liegt es nahe, diese Bedingung in eine RCC zu übersetzen. Der Primärschlüssel würde dann zum Erzeuger am Anfang der RCC und der Fremdschlüssel würde zu seinem Nachfolger, also dem Konsumenten am Ende der RCC.

Folgerung 7: Sei k ein Fremdschlüssel und e der zugehörige Primärschlüssel. Dann gilt für eine RCC e→k die RCC-Teilmengen-Annahme.

Begründung: Bedingt durch die referentielle Integrität gilt, dass für jeden Wert $\varphi \in V_k$ auch $\varphi \in V_e$ gilt. Demzufolge ist der Wertebereich des Fremdschlüssels/Konsumenten stets eine Teilmenge des Wertebereichs des Primärschlüssels/Erzeugers. ■

Wir können die RCC-Teilmengen-Annahme also auch so formulieren, dass wir davon ausgehen, dass zwischen Konsument und Erzeuger eine Fremdschlüssel-Beziehung besteht.

Unter der RCC-Teilmengen-Annahme ergeben sich zwei Fälle. Im ersten Fall ist $c_e \leq c_k$, also gilt $V_e \subseteq V_k$. Im zweiten Fall ist $c_e > c_k$, also gilt $V_e \supset V_k$.

Betrachten wir zunächst den ersten Fall. Aus $V_e \subseteq V_k$ folgt, dass jeder ankommende Wert im Backend des Konsumenten vorhanden ist. Damit wird insbesondere jeder in e geladene Wert in k induziert.

Im zweiten Fall, d. h. bei $c_k < c_e$ ergibt sich dann, dass die DBD des Konsumenten eine Teilmenge der DBD des Erzeugers ist. Dadurch kann nicht mehr angenommen werden, dass die Menge der ankommenden Werte eine Teilmenge der DBD des Konsumenten ist. Es gibt also mindestens einen Wert aus der DBD von e, der nicht in der DBD des Konsumenten liegt. Nur die Werte, die in e geladen sind und in der DBD von k liegen, sind nach dem Füllzyklus umgebungsvollständig. Allerdings liegt in diesem Fall nur eine minimale Anzahl von Werten nicht in der DBD des Konsumenten. Insofern können wir die Annahme als pessimistisch betrachten, da hierdurch der Füllstand einer Cache-Tabelle jeweils maximal wird. Mit anderen

Worten: die Annahme führt in jedem Fall dazu, dass die Werte n_T für alle Cache-Tabellen T größer sind als ohne die Annahme.

2.8.3 Überdeckung zwischen zwei Spalten

Wie wir in Abschnitt 1.5 gesehen haben, müssen alle Werte, die in einem Vorgänger geladen sind und im Backend vorhanden sind, nach dem Füllzyklus umgebungsvollständig sein.

Maßgeblich bestimmend für die Wahrscheinlichkeit, dass ein ankommender Wert nach dem Füllzyklus im Konsument umgebungsvollständig ist, ist in diesem Zusammenhang die Überdeckung zwischen dem betrachteten Konsumenten und seinem Vorgänger.

Definition: Seien e und k zwei Spalten. Die Größe

$$ü_{k,e} = | V_k \cap V_e | / c_e \qquad (2.8.3.1)$$

bezeichnen wir als **Überdeckung** der DBD der Spalte e zur DBD der Spalte k (oder auch kurz Überdeckung von e und k).

Die Überdeckung sagt uns also, wie groß der (durchschnittliche) Anteil der im Erzeuger e geladenen Werte (bei gleichmäßiger Verteilung dieser Werte über die DBD von e) ist, die im Konsument k über die RCC $e \rightarrow k$ induziert werden.

Man beachte, dass die Überdeckung zwischen zwei Spalten nicht symmetrisch ist, d. h. es gilt nicht $ü_{k,e} = ü_{e,k}$ (zumindest nicht im Allgemeinen).

Folgerung 8: Seien e und k zwei Spalten mit $e \rightarrow k$. Unter der RCC-Teilmengen-Annahme können wir die Größe der Schnittmenge abschätzen durch:

$$| V_e \cap V_k | = \min(c_e, c_k). \qquad (2.8.3.2)$$

Somit gilt für die Überdeckung:

$$ü_{k,e} = \min(c_e, c_k)/c_e = \min(c_e / c_e, c_k / c_e) = \min(1, c_k / c_e). \qquad (2.8.3.3)$$

Begründung: Nach der RCC-Teilmengen-Annahme ist die kleinere Menge immer eine Teilmenge der größeren Menge. Die Schnittmenge der DBDs ist also identisch zur kleineren der beiden DBDs und somit kann auch die Mächtigkeit berechnet werden. ■

Mit Hilfe der kombinierten Zugehörigkeitsfunktion können wir die Überdeckung zweier Spalten explizit berechnen, ohne die RCC-Teilmengen-Annahme zu verwenden. Wir iterieren dabei einfach über die Menge der Werte in der DBD von e und zählen, wie oft die kombinierte Zugehörigkeitsfunktion von k und e gleich 1 wird. Damit erhalten wir die Mächtigkeit der Schnittmenge zwischen den DBDs von e und k, die wir dann nur noch durch die Kardinalität von e teilen müssen.

2.8.4 Anzahl der über eine RCC induzierte Werte

Mit unserem bisher gewonnen Wissen können wir die Anzahl $v_{i|e\rightarrow k}$ der Werte abschätzen, die über eine RCC e→k in einen Konsumenten k insgesamt induziert werden. D. h. wir betrachten die Situation nach dem Füllzyklus.

Die in diesem Abschnitt vorgeschlagene Abschätzung der Anzahl wird unter anderem im Modell zur Berechnung der Füllschritte in baumartigen Cache-Groups (d. h. Cache-Groups, deren SBG ein Baum und deren einziger Cache-Key der Wurzel-Knoten ist; siehe Kapitel 3) verwendet. Außerdem wird in diesem Abschnitt die Rolle der Überdeckung zweier Spalten deutlich. Grundsätzlich kann zudem die Abschätzung dieser Anzahl zur Herleitung der Abschätzung der insgesamt (über verschiedene RCCs) induzierten Werte als Kombination von Ereignissen verwendet werden.

Man beachte, dass man die Größe $v_{i|e\rightarrow k}$ exakt bestimmen kann, wenn man genau weiß, welche Werte über die betrachtete RCC ankommen und welche Werte in der DBD einer Spalte liegen. Eine derartig genaue Untersuchung ist allerdings auf Grund ihrer Komplexität nicht möglich, weshalb wir in unseren Modellen davon abstrahieren. Im Rahmen unserer Modelle kennen wir allgemein nur die abgeschätzte Mächtigkeiten der verschiedenen Wertemengen. Aus der abgeschätzten Mächtigkeit der ankommenden Werte kann dann die Mächtigkeit der geladenen Werte abgeschätzt werden.

Wir wollen den Sachverhalt etwas formaler betrachten. Die nächste Folgerung beschreibt, wie man die Anzahl der über eine bestimmte RCC induzierten Werte abschätzen kann.

Folgerung 9: Für e→k gilt:

$$\max(0,\ v_{l|e} + \ddot{u}_{k,e}\cdot c_e - c_e) \leq v_{i|e\rightarrow k} \leq \min(v_{l|e},\ \ddot{u}_{k,e}\cdot c_e). \qquad (2.8.4.1)$$

Unter der Annahme, dass die Wahrscheinlichkeit für jeden Wert gleich hoch ist in e geladen zu sein, können wir im Sinne einer Average-Case-Analyse die Anzahl der über die RCC induzierten Werte wie folgt annähern:

$$v_{i|e\rightarrow k} = v_{l|e}\cdot\ddot{u}_{k,e}. \qquad (2.8.4.2)$$

Beweis: Wir wissen nun zum einen, dass ein Wert aus der DBD von e nur dann über die RCC e→k induziert sein kann, wenn er in e geladen ist. Zum anderen wissen wir, dass ein Wert aus der DBD von e ebenfalls nur dann über die RCC e→k induziert sein kann, wenn er in der DBD von k liegt. Ein über e→k induzierter Wert liegt also in der Schnittmenge zwischen DBD von e, DBD von k und der Menge der in e geladenen Werte. Es gilt also:

$$V_{i|e\rightarrow k} = V_{l|e} \cap V_k \cap V_e.$$

Die obere Schranke erhalten wir nun, indem wir annehmen, dass alle geladenen Werte in der Schnittmenge der DBDs zwischen e und k liegen. Die Anzahl wird dann aber entweder durch die Mächtigkeit der Schnittmenge oder der Mächtigkeit der in e geladenen Werte nach oben beschränkt.

Die untere Schranke erhalten wir, indem wir annehmen, dass so viele in e geladene Werte wie möglich nicht in der Schnittmenge liegen. Ist die Anzahl der in e geladenen Werte allerdings

größer als die Differenzmenge zwischen den DBDs von e und k, so muss ein Anteil dieser Werte in der Schnittmenge liegen.

Betrachten wir nun den Fall der Average-Case-Analyse. Für die Menge der in e geladen Werte und der Schnittmenge zwischen der DBD von e und der DBD von k können wir eine gemeinsame Obermenge bestimmen. Beide Mengen sind Teilmengen der DBD von e. Wir können also schreiben:

$$V_e \cap V_k \subseteq V_e \quad \text{und} \quad V_{l|e} \subseteq V_e.$$

Unter Verwendung von Formel 2.5.1 können wir die Mächtigkeit der Schnittmenge zweier Mengen A und B, die in einer gemeinsamen Obermenge C liegen, wie folgt abschätzen:

$$|A \cap B| \approx |A| \cdot |B| / |C|.$$

Wir können dieses Rechenschema nun auf die betrachtete Menge anwenden und somit die Mächtigkeit der Menge der über e→k induzierten Werte abschätzen:

$$v_{i|e\to k} = |V_{l|e}| \cap |V_k \cap V_e| = v_{l|e} \cdot (\ddot{u}_{k,e} \cdot c_e) / c_e = v_{l|e} \cdot \ddot{u}_{k,e}. \blacksquare \qquad (2.8.4.1)$$

2.8.5 Wertemengen-Pfade

In diesem Abschnitt wollen wir wichtige Teilgraphen aus zusammenhängenden Kanten des SBG betrachten. Wir untersuchen in den nächsten Abschnitten Wertemengen-Pfade, d. h. zusammenhängende Wege von RCC-Kanten im SBG.

Definition: Ein zusammenhängender, zyklenfreier Weg von RCC-Kanten in einem SBG heißt **Wertemengen-Pfad** (**WMP**). Ein WMP heißt **maximal**, wenn er an einem Basiserzeuger beginnt.

WMPe haben einen großen Einfluss auf die Mengen geladener Werte in Spalten. Sie bilden sozusagen einen Pfad, auf dem die Wertemengen „wandern“.

Bemerkung: Wir können WMPe als Spezialfälle allgemeiner Teilgraphen des SBG betrachten. Diese Teilgraphen sind Bäume und bestehen lediglich aus RCC-Kanten und den damit verbunden Knoten (Spalten). Bei baumartigen Cache-Groups spielen diese so genannten RCC-Teilbäume eine wichtige Rolle für die Berechnung der Füllstände mit Hilfe von Histogrammen.

Transitive Erweiterung der RCC-Implikation

WMPe erweitern gewissermaßen die RCC-Implikation für einzelne Werte transitiv: Ein Wert, der in den DBDs aller Spalten entlang alles WMP vorhanden und in der Spalte am Anfang des WMPs geladen ist, wird in die Spalten auf dem WMP (insbesondere auch in die Spalte am Ende des WMP) induziert.

Folgerung 10: Sei $c_1 \to \ldots \to c_n$ ein WMP und $\varphi \in V_{l|c_1}$ ein in c_1 geladener Wert. Dann liegt φ nach dem Füllzyklus in der Menge $V_{i|c_n}$ der in c_n induzierten Werte:

$$\chi_{c_2 \cdot \ldots c_n}(\varphi) = 1 \Rightarrow \varphi \in V_{i|c_n} \qquad (2.8.5.1)$$

Begründung: Auf Grund der RCC-Implikation gilt:

$$\varphi \in V_{l|c_i} \Leftarrow \chi_{c_i}(\varphi) = 1 \wedge \varphi \in V_{l|c_{i-1}} \quad \text{für alle } 1 < i \leq n.$$

Aufgrund der Definition der kombinierten Zugehörigkeitsfunktion für ein Paar von Spalten kann nun die Voraussetzung umgeschrieben werden:

$$\chi_{c_i}(\varphi) = 1 \wedge \varphi \in V_{l|c_{i-1}} \Leftarrow \chi_{c_i}(\varphi) = 1 \wedge \chi_{c_{i-1}}(\varphi) = 1 \wedge \varphi \in V_{l|c_{i-2}} \Leftrightarrow \chi_{c_i \cdot c_{i-1}}(\varphi) = 1 \wedge \varphi \in V_{l|c_{i-2}}.$$

Für die Spalte am Anfang des WMPs, also für i = 1 ist die Voraussetzung auf der rechten Seite erfüllt. Durch Induktion gelangt man zur Formulierung der ursprünglichen Folgerung. ■

Definition: Ein Wert φ heißt **über den WMP** $c_1 \rightarrow \ldots \rightarrow c_n$ **induziert**, wenn er in c_1 geladen und in den DBDs aller Spalten $c_1, \ldots, c_n$ vorhanden ist:

$$\varphi \in V_{i|c_1 \rightarrow \ldots \rightarrow c_n} \Leftrightarrow \varphi \in V_{l|c_1} \cap V_{c_2} \cap \ldots \cap V_{c_n}. \qquad (2.8.5.2)$$

Aus der Definition ergibt sich, dass ein über den WMP $c_1 \rightarrow \ldots \rightarrow c_n$ induzierter Wert in allen Spalten auf dem WMP geladen ist.

Maximale Wertemengen-Pfade und Basiserzeuger

Mit Hilfe von maximalen WMPen kann die Menge der Basiserzeuger, die zu einer Spalte gehören, gefunden werden.

Definitionen: Ein Basiserzeuger b heißt **Basiserzeuger einer Spalte** c, wenn zwischen b und c ein WMP (eventuell der Länge null) liegt.[18] Die Spalte b heißt dann auch **Basiserzeuger der Cache-Tabelle** T(c).

Zu einer gegebenen Spalte können mehrere Basiserzeuger existieren. Als Beispiel schauen wir uns folgenden Ausschnitt aus einer Cache-Group an. Im Ausschnitt liegen drei Spalten a, b und c in jeweils unterschiedlichen Cache-Tabellen. In der Cache-Group existieren zwei RCCs: a→c und b→c. Die Spalten a und b seien am Ende einer Schmuggler-Beziehung, aber selbst keine Konsumenten. Dementsprechend sind beide Spalten Basiserzeuger (von c). Abbildung 7 zeigt den beschriebenen Ausschnitt aus der Cache-Group.

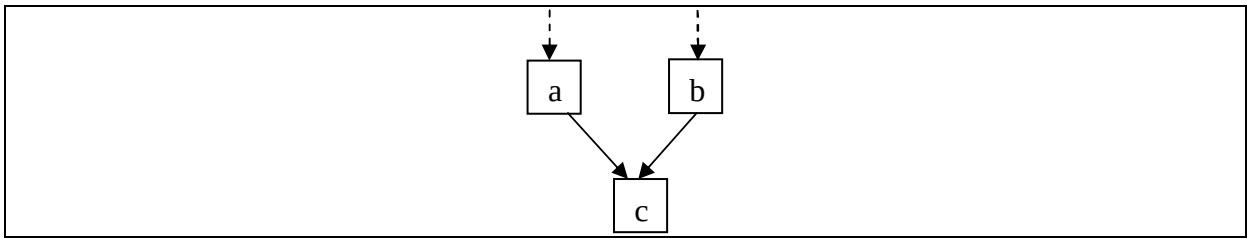

Abbildung 7: Ausschnitt aus einer Cache-Group

Isolierte Wertemengen-Pfade

Die Objekte, die wir im Folgenden betrachten, sind Spezialfälle von WMPen, bei denen alle beteiligten RCCs isoliert sind. Solche WMPe haben angenehme Eigenschaften, z. B. ist durch

[18] Dieser WMP ist dann maximal.

sie der Basiserzeuger der Konsumenten auf dem WMP eindeutig bestimmt. Ankommende Werte können also nur aus diesem Basiserzeuger stammen.

Definition: Ein WMP heißt **isoliert**, wenn alle beteiligten RCCs isoliert sind.

Diese isolierten WMPe spielen in baumartigen Cache-Groups (d. h. Cache-Groups, deren SBG ein Baum und deren einziger Cache-Key der Wurzel-Knoten ist) eine große Rolle. Alle RCCs und damit auch alle WMPe einer baumartigen Cache-Group sind isoliert, denn jede Cache-Tabelle hat nur einen Konsumenten. Dieser einzelne Konsument hat nur eine eingehende RCC und kann natürlich auch kein Schmuggelziel sein.

Folgerung 11: Für jeden Konsumenten k auf einem isolierten WMP ist der Basiserzeuger eindeutig definiert.

Beweis: Sei k ein beliebiger Konsument auf einem isolierten WMP, d. h. bei k endet genau eine RCC, die Teilweg eines isolierten WMPs ist. Da zu k genau eine RCC führt, gibt es jeweils auch nur eine Möglichkeit, den WMP vom Konsumenten aus rückwärts zu verfolgen. Entweder wir gelangen dadurch zum Basiserzeuger oder zu einem weiteren Konsumenten k', für den wieder genau eine eingehende RCC existiert. In diesem Fall können wir den WMP weiter rückwärts verfolgen usw. Wir gelangen dann irgendwann zwangsläufig zum eindeutigen Basiserzeuger. ■

Da zu jedem Konsumenten der zugehörige Basiserzeuger eindeutig ist, können wir eine Zuordnungsfunktion definieren. Diese Funktion liefert zu jedem Konsument k den entsprechenden Basiserzeuger. Formal schreiben wir:

Definition: Sei c eine Spalte auf einem isolierten WMP Cache-Group. Die **Basiserzeuger-Zuordnungsfunktion** b(c) liefert einen eindeutig bestimmten Basiserzeuger zu c.

Abbildung 8 zeigt einen Algorithmus, mit dessen Hilfe man zu einer gegebenen Spalte den zugehörigen Basiserzeuger finden kann.

```
Spalte finde_Basiserzeuger(Spalte k) {
      while (hat_Vorgänger(k)) {
            k = finde_Vorgänger(k);
      }
      // Kein weiterer Vorgänger -> wir haben den Basiserzeuger gefunden.
      return k;
}
```

Abbildung 8: Algorithmus, um zu einer gegebenen Spalte den Basiserzeuger zu finden.

Analog zu isolierten RCCs in Abschnitt 2.8.1 lässt sich für isolierte WMPe feststellen, woher ein geladener Wert stammt. Da der Basiserzeuger eindeutig bestimmt ist, können die Werte nur aus dieser Spalte stammen. Wir können deshalb die nächste Folgerung formulieren:

Folgerung 12: Sei $c_1 \rightarrow \ldots \rightarrow c_n$ ein isolierter WMP und $\varphi \in V_{l|c_1}$. Dann gilt:

$$\varphi \in V_{vc|c_n} \Leftrightarrow \chi_{c_2 \cdot \ldots c_n} = 1, \quad V_{l|c_1} \supseteq \ldots \supseteq V_{l|c_n} \quad \text{und} \quad |V_{l|c_1}| \geq \ldots \geq |V_{l|c_n}| \qquad (2.8.5.2)$$

Beweis: Die Implikation von rechts nach links gilt für alle WMPe nach Folgerung 10. Dementsprechend gilt diese Richtung der Implikation insbesondere für isolierte WMPe. Da bei c_n ausschließlich die RCC $c_{n-1} \rightarrow c_n$ und sonst keine Kante im SBG (auch keine Schmuggler-Kante) endet, lässt sich nun zeigen, dass der in c_n umgebungsvollständige – also insbesondere geladene Wert – nur über die isolierte RCC am Ende des betrachteten WMPs angekommen sein kann. Für die vorletzte Spalte gilt entsprechendes und so kann der WMP rückwärts verfolgt werden, um zu begründen, dass der Wert ursprünglich in c_1 geladen sein muss. Der Wert kann aber nur ankommen, wenn er in allen DBDs der Spalten auf dem WMP vorhanden ist. Ansonsten wäre der Wert in der betreffenden Spalte nicht geladen und hätte nicht zu einem Laden in seinen Nachfolgern geführt. Der Wert muss also in den DBDs aller Spalten auf dem WMP sein und somit in der Schnittmenge aller DBD liegen, womit die kombinierte Zugehörigkeitsfunktion gleich 1 wird. Damit gilt die Implikation auch von links nach rechts.

Aufgrund der Tatsache, dass ein geladener Wert nur über eine einzige eingehende RCC angekommen sein kann, ergibt sich, dass die Mengen der geladenen Werte entlang des WMPs einander einschließen: φ muss in allen Spalten zwischen c_1 und c_2 (inklusive c_1) geladen sein. Dies gilt für jeden Wert aus der Menge der in c_i geladenen Werte und für alle i zwischen 1 und n, also allen Spalten auf dem WMP. Somit ist die betrachtete Menge der in eine Spalte auf dem WMP geladenen Werte eine Teilmenge der Mengen der in ihren Vorgängern geladenen Werte.

Aus der Teilmengen-Eigenschaft ergibt sich, dass auch die Mächtigkeiten der Mengen der geladenen Werte nicht zunehmen können, je weiter man den WMP vom Anfang zum Ende verfolgt. ■

Ladewahrscheinlichkeit eines Werts in Konsumenten auf einem isolierten WMP

Die in Spalten auf dem WMP geladenen Werte müssen aus einem Basiserzeuger am Anfang des WMPs stammen. Die Wahrscheinlichkeit, dass ein Wert in eine Spalte geladen wird hängt also von der Wahrscheinlichkeit ab, ob ein Wert in den entsprechenden Basiserzeuger (am Anfang des WMPs) geladen wurde.

So kann der Füllstand der Cache-Tabellen z. B. in baumartigen Cache-Groups (aber auch allgemein in Cache-Tabellen entlang eines isolierten WMPs) mit Hilfe von Histogrammen abgeschätzt werden, da jeder Konsument am Ende eines isolierten WMPs liegt. Wir betrachten also im Folgenden einen Konsumenten k in einer stabilen Cache-Group auf einem isolierten WMP und gehen dabei davon aus, dass k kein Cache-Key ist.

Prinzipiell behalten unsere Überlegungen auch für Konsumenten, die gleichzeitig Cache-Key sind, ihre Gültigkeit. Jedoch spielt es dann eine Rolle, ob der durch die Benutzeranfrage referenzierte und den betrachteten Füllzyklus auslösende Wert in k oder einem anderen Cache-Key referenziert wurde. Des Weiteren können wir relativ einfach unsere Überlegungen auch auf Zeitpunkte vor dem Ende des Füllzyklus übertragen, indem wir die Semantik der Wahrscheinlichkeiten verändern.

Die Abschätzung des Füllstands der Cache-Tabelle T(k) geschieht im Rahmen einer Average-Case-Betrachtung. Wir nehmen zunächst an, wir haben eine Wahrscheinlichkeitsverteilung gegeben, die uns sagt, wie wahrscheinlich es für einen bestimmten Wert ist, zur Menge der in k geladenen Werte zu gehören. Haben wir eine solche Wahrscheinlichkeitsverteilung, so kann

mit Hilfe des Histogramms des Konsumenten die Anzahl der zu ladenden Sätze einfach abgeschätzt werden.

Wir definieren Zufallsvariablen X_φ für jeden Wert φ aus der DBD des Konsumenten k. Diese Zufallsvariablen sagen uns, wie viele Sätze aus dem Backend für den Werts φ in die Cache-Tabelle geladen wurden:

$$X_\varphi := \begin{cases} h_k(\varphi) \text{ wenn } \varphi \in V_{i|k} \\ 0 \text{ sonst} \end{cases}$$

Die Zufallsvariable gibt das Verhalten des Konsumenten korrekt wieder, denn es können nur Werte in ihn induziert, nicht aber geschmuggelt werden; das liegt daran, dass der dort endende WMP und insbesondere somit auch die dort endende RCC isoliert ist: Bei k endet keine weitere RCC und vor allem auch keine Schmuggler-Kante.

Den Erwartungswert dieser Zufallsvariable können wir mit dem Produkt aus Häufigkeit $h_k(\varphi)$ von φ in k und der Wahrscheinlichkeit $p_{i|k}(\varphi)$, dass φ in k induziert ist, bestimmen (zur Erläuterung der Notation sei auf Anhang B verwiesen):

$$E(X_\varphi) = h_k(\varphi) \cdot p_{i|k}(\varphi) = h_k(\varphi) \cdot p_{i|k}(\varphi) =: h_{i|k}(\varphi).$$

Um nun die Anzahl der in eine Cache-Tabelle geladenen Werte zu bestimmen, definieren wir eine weitere Zufallsvariable Z, die die Anzahl der induzierten Sätze zählt:

$$Z = \sum_{\varphi \in V_k} X_\varphi \, .$$

Im Sinne einer Average-Case-Analyse können wir nun die Anzahl der in T(k) geladenen Sätze, also die Mächtigkeit der Werteumgebung der induzierten Werte, durch den Erwartungswert von Z annähern. Wir schreiben:

$$s_{i|k} = E(Z) = E\left(\sum_{\varphi \in V_k} X_\varphi\right) = \sum_{\varphi \in V_k} E(X_\varphi) = \sum_{\varphi \in V_k} p_{l|k}(\varphi) h_{i|k}(\varphi) \, . \qquad (2.8.5.3)$$

Betrachten wir nun, wie man die Wahrscheinlichkeitsverteilung der Wahrscheinlichkeit $p(\varphi \in V_{i|k}) := p_{l|k}(\varphi)$ näherungsweise bestimmen kann.

Definition: Wir nennen die Wahrscheinlichkeit, dass ein Wert φ in einer Spalte c geladen ist **Ladewahrscheinlichkeit** (von φ in c).

Bemerkung: Dieser Begriff gilt allgemein für alle Spalten, nicht nur für Spalten auf einem isolierten WMP.

Sicherlich hängt die Ladewahrscheinlichkeit in einem Konsumenten auf einem isolierten WMP von der Ladewahrscheinlichkeit in seinem Basiserzeuger ab. Nur wenn ein Wert im Basiserzeuger geladen ist, kann er über den WMP beim Konsumenten ankommen.

Außerdem hat der WMP zwischen Basiserzeuger und Konsument einen großen Einfluss auf die Ladewahrscheinlichkeit im Konsumenten. Ist der Wert in der DBD einer der Spalten auf dem WMP nicht vorhanden, so wird er dort nicht geladen und kann von dort aus nicht das Laden des Werts in nachfolgenden Spalten verursachen. Der Wert kann in einer solchen Situation also nicht beim betrachteten Konsumenten ankommen, womit die Ladewahrscheinlichkeit gleich null ist.

Folgerung 13: Sei c eine Spalte auf einem isolierten WMP und $\varphi \in V_c$ ein Wert aus der DBD von c. Dann gilt für die Ladewahrscheinlichkeit von φ in c folgender Zusammenhang:

$$p_{l|c}(\varphi) = p_{l|b(c)}(\varphi) \cdot \chi_{b(c)\cdot\ldots\cdot c}(\varphi). \qquad (2.8.5.4)$$

Beweis: Aus Folgerung 12 wissen wir, dass ein Wert in einer Spalte am Ende eines isolierten WMPs geladen ist, wenn er in der Schnittmenge der DBDs aller Spalten auf dem WMP und in der Menge der in der Spalte am Anfang des WMPs geladenen Werte liegt. Nach Voraussetzung ist der WMP zwischen Basiserzeuger und betrachtetem Konsumenten isoliert. Wir können nun zwei Fälle unterscheiden.

Sei $\chi_{b(c)\cdot\ldots\cdot c}(\varphi) = 0$. Dann ist φ in mindestens einer der DBDs der Spalten auf dem WMP nicht vorhanden. In dieser Spalte würde der Wert nicht geladen und könnte damit auch nicht beim Konsumenten ankommen. Anders ausgedrückt: die Wahrscheinlichkeit, dass φ in c geladen ist, ist gleich null. Es gilt also $p_{l|c}(\varphi) = 0$. Die Formel bringt das gleiche Ergebnis durch Einsetzen:

$$p_{l|c}(\varphi) = p_{l|b(c)}(\varphi) \cdot \chi_{b(c)\cdot\ldots\cdot c}(\varphi) = p_{l|b(c)}(\varphi) \cdot 0 = 0.$$

Sei $\chi_{b(c)\cdot\ldots\cdot c}(\varphi) = 1$. Dann ist der Wert in den DBDs aller Spalten auf dem WMP vorhanden und wird in allen Spalten geladen, wenn er im Basiserzeuger geladen ist. Da der einzige Weg, wie φ in den Konsumenten geladen werden kann, der WMP ist, ist die Ladewahrscheinlichkeit im Konsumenten gleich der Ladewahrscheinlichkeit im Basiserzeuger. Es gilt in diesem Fall also: $p_{l|c}(\varphi) = p_{l|b(c)}$. Auf dieses Ergebnis kommen wir auch durch Einsetzen:

$$p_{l|c}(\varphi) = p_{l|b(c)}(\varphi) \cdot \chi_{b(c)\cdot\ldots\cdot c}(\varphi) = p_{l|b(c)}(\varphi) \cdot 1 = p_{l|b(c)}(\varphi). \blacksquare$$

Generell (also bei nicht-isolierten WMPen) können wir nicht wissen, ob ein Wert dadurch in den Basiserzeuger durch Schmuggeln oder Referenzieren (in einer Benutzeranfrage) in den Basiserzeuger eingeführt wurde. Ausschließlich diese beiden Möglichkeiten existieren, denn der Wert kann unmöglich in den Basiserzeuger induziert worden sein: Ansonsten gäbe es eine entsprechende eingehende RCC und der Basiserzeuger wäre ein Konsument, was ein Widerspruch wäre.

Wir können die Ladewahrscheinlichkeit im Basiserzeuger als Wahrscheinlichkeit kombinierter Ereignisse betrachten. Ein Wert kann allerdings nur referenziert oder geschmuggelt worden sein: Entweder der Wert ist der referenzierte Cache-Key-Wert in der betrachtenden Benutzeranfrage, dann wird er nicht mehr in die Spalte geschmuggelt, denn nach dem ersten Füllschritt ist er bereits umgebungsvollständig. Gleiches gilt natürlich auch umgekehrt. Die Gesamtwahrscheinlichkeit ergibt sich also aus der Summe aller Teilwahrscheinlichkeiten:

$$p_{l|b(k)}(\varphi) = p_{ref|b(k)}(\varphi) + p_{s|b(k)}(\varphi). \qquad (2.8.5.5)$$

2.9 Schmuggler-Beziehungen

Neben RCCs haben Schmuggler-Beziehungen ebenfalls einen großen Einfluss auf das Füllverhalten von Cache-Groups. Auf der Ebene einzelner Werte ist die Wahrscheinlichkeit $p_{s|z}(\varphi)$, dass ein Wert φ in ein Schmuggelziel z geschmuggelt wurde, an die Wahrscheinlichkeit $p_{i|k}(\varphi)$ gekoppelt, dass φ in einer Spalte k auf einem bei z abgehenden isolierten WMP

induziert wird. Auf der Ebene der *Menge* der in ein Schmuggelziel geschmuggelter Werte steigt mit größer werdender Anzahl auch die Anzahl der „wandernden“ Werte, die an nachfolgenden Spalten ankommen. Diese erhöhte Anzahl führt dann wiederum zu größeren Füllständen.

Prinzipiell werden Werte in eine Spalte geschmuggelt, wenn in eine Schmuggelquelle ungleich des Schmuggelziels Werte induziert werden. Durch das Induzieren (in eine andere Schmuggelquelle) werden Sätze in die Cache-Tabelle geladen, die auch in der Werteumgebung eines geschmuggelten Wertes im Schmuggelziel liegen.

Betrachten wir nun die Menge der Sätze, die dafür verantwortlich ist, dass Werte in das Schmuggelziel geschmuggelt werden.

Definition: Wir nennen die Menge der während Zeitpunkt τ in die Cache-Tabelle T(z) eines Schmuggelziels z geladenen und in der Werteumgebung eines in z geschmuggelten Werts liegen **Schmuggelumgebung** $\tilde{S}_{s|z|\tau}$ von z zum Zeitpunkt τ.

Die Schmuggelumgebung von z ist im Allgemeinen nicht gleich der Werteumgebung aller in z geschmuggelten Werte. Letztere Satzmenge umfasst auch diejenigen Sätze, die zwar im Backend vorhanden sind, aber nicht geladen wurden. Ansonsten müssten alle geschmuggelten Werte umgebungsvollständig sein.

Wie wir wissen, ist die Schmuggelumgebung des Schmuggelziels z zum Zeitpunkt τ leer, wenn z zu diesem Zeitpunkt aktiv war:

$$\tilde{S}_{s|z|\tau} = \begin{cases} \varnothing \text{ falls } z = c \\ S'_{i|c|\tau} \text{ sonst} \end{cases}, \qquad s := \tilde{s}_{s|z|\tau} = \begin{cases} 0 \text{ falls } z = c \\ s'_{i|c|\tau} \text{ sonst} \end{cases}$$

Begründung: War im betrachteten Füllschritt z aktiv (d. h. es gilt z = c), dann wurden Werte in z induziert. In diesem Fall wurden aber keine Werte während dieses Füllschritts in z geschmuggelt, da keine Spalte mit sich selbst in einer Schmuggler-Beziehung stehen kann. War aber z nicht die aktive Spalte, so tragen alle während des Füllschritts geladenen Sätze zum Schmuggeln von Werten bei, liegen also in der Schmuggelumgebung. Diese Sätze wurden aber deswegen geladen, weil ein entsprechender Wert in c induziert wurde. D. h. die Sätze liegen jeweils in einer Werteumgebung eines in c während des betrachteten Füllschritts induzierten Werts. Somit ist die Schmuggelumgebung der Werteumgebung der während des Füllschritts in c induzierten Werte. Die Abschätzung der Mächtigkeit der Schmuggelumgebung ergibt sich offensichtlich.

Wie wir in Abschnitt gesehen haben, ist die Wahrscheinlichkeit für einen Wert φ in einem Konsumenten k induziert zu sein unter anderem von der Schmuggelwahrscheinlichkeit von φ im Basiserzeuger b(k) abhängig, wenn k an einem isolierten WMP liegt und sein Basiserzeuger ein Schmuggelziel ist. Die Schmuggelwahrscheinlichkeit fließt also direkt in Formel 2.8.5.4 ein. Außerdem ist offensichtlich, dass die Wahrscheinlichkeit dafür, dass ein Wert φ in ein Schmuggelziel z geschmuggelt wurde, die Wahrscheinlichkeit dafür, dass ein Wert in einen Nachfolger von z induziert wird, beeinflusst. Diese letztere Wahrscheinlichkeit beeinflusst wiederum die Induzierungswahrscheinlichkeit in dessen Nachfolgern usw.

Wir können drei Einflussfaktoren auf die Schmuggelwahrscheinlichkeit identifizieren:

- Die Mächtigkeit der Schmuggelumgebung (und damit implizit auch der Zeitpunkt τ des betrachteten Füllschritts,
- die Häufigkeit des Werts und
- die Kardinalität der Backend-Tabelle

Die Abhängigkeit der Schmuggelwahrscheinlichkeit von der Häufigkeit $h_z(\varphi)$ des betrachteten Werts φ im Schmuggelziel z erklärt sich folgendermaßen: Wenn φ häufiger in z vorkommt, dann ist es wahrscheinlicher, dass ein Satz η aus der Werteumgebung $S_{z,\varphi}$ in der Schmuggelumgebung $\tilde{S}_{s|z|\tau}$ liegt. Gleiches gilt natürlich auch umgekehrt. Mit steigender Kardinalität der Backend-Tabelle sinkt jedoch die Schmuggelwahrscheinlichkeit unter der Annahme, dass jeder Satz $\eta' \in S_{T(z)}$ aus der Backend-Tabelle gleich wahrscheinlich geladen wird, denn dann ist die Wahrscheinlichkeit geringer, dass einer der geladenen Sätze in $S_{z,\varphi}$ liegt.

Schreibweise: Wir schreiben für die **Schmuggelwahrscheinlichkeit** $p_{s|z}(\varphi) =: w_z(h, s)$. Dabei ist $h = h_z(\varphi)$ die Häufigkeit von φ in Schmuggelziel z und s die abgeschätzte Mächtigkeit der Schmuggelumgebung von z.

Nach dieser theoretischen Einführung wollen wir uns nun mit der Berechnung der Schmuggelwahrscheinlichkeit beschäftigen. Wir führen aufgrund ihrer Länglichkeit an dieser Stelle keine Beweise auf, sondern verweisen hierfür auf den Anhang C.

Wir können die Schmuggelwahrscheinlichkeit eines Werts mit Häufigkeit h in Spalte z unter der Voraussetzung, dass s die Mächtigkeit der Schmuggelumgebung abschätzt wie folgt berechnen:

$$w_z(h,s) = 1 - \frac{\binom{|S_{T(z)}| - h}{s}}{\binom{|S_{T(z)}|}{s}} = 1 - \frac{(|S_{T(z)}| - h)! \cdot (|S_{T(z)}| - s)!}{(|S_{T(z)}| - h - s)! \cdot |S_{T(z)}|!} \qquad (2.8.1)$$

Zur Herleitung der Berechnungsvorschrift verweisen wir auf Anhang C.

Diese Formel ist mit dem Rechner schwer zu handhaben, da sie einerseits nur für ganze Zahlen anwendbar ist und andererseits in den Binomialkoeffizienten versteckt viele Fakultäten enthält. Da wir in den in dieser Arbeit vorgestellten Modellen üblicherweise mit Annäherungen, Durchschnittswerten oder Erwartungswerten rechnen sind die Einflussgrößen auf die Schmuggelwahrscheinlichkeit (d. h. insbesondere $s_{s|z|\tau}$) keine ganzen Zahlen. Aus diesem Grund wollen wir des Weiteren eine Fortsetzung der Formel auf reelle Zahlen finden. Um dieser Problematik entgegen zu wirken, werden wir zwei Möglichkeiten betrachten:

- Annäherung mit Hilfe eine Stirling-Formel und
- rekursive Definition und Interpolation.

Betrachten wir die Annäherung mit Hilfe der Stirling-Formel. Die Stirling-Formeln existieren in unterschiedlichen Versionen. Nach der einfachsten Stirling-Formel gilt [Föl04]:

$$\ln x! \approx x \cdot \ln x.$$

Die Stirling-Formel hat den Vorteil, dass sie nicht nur auf ganzen Zahlen definiert ist wie die Fakultät, sondern auch für reelle Zahlen angewendet werden kann. Sie stellt so gesehen die Fortsetzung der Fakultät auf reelle Zahlen in einer Näherung dar.

Mit Hilfe des gewonnenen Wissens können wir nun die Stirling-Formel benutzen, um einzelne Fakultäten und damit die gesamte Schmuggelwahrscheinlichkeit anzunähern.

Folgerung 14: Die Schmuggelwahrscheinlichkeit kann mit Hilfe der Stirling-Formel wie folgt angenähert werden:

$$w_z(h, s) \approx 1 - \frac{(|S_{T(z)}| - h) \cdot (|S_{T(z)}| - s)}{(|S_{T(z)}| - h - s) \cdot |S_{T(z)}|} \qquad (2.8.2)$$

Beweis: Siehe Anhang C ■

Eine andere Möglichkeit, der Problematik entgegenzuwirken, dass die ursprüngliche Form der Berechnungsvorschrift für die Schmuggelwahrscheinlichkeit mit dem Rechner nur schwer zu handhaben ist, kann darin gesehen werden, die Formel rekursiv darzustellen. Durch die rekursive Darstellung kann die Formel iterativ berechnet werden. Dabei gibt es im Gegensatz zur direkten Berechnung über die Darstellung mit Fakultäten keine Überläufe.

Wir wollen die in Formel 2.8.1 dargestellte Form der Schmuggelwahrscheinlichkeit weiter zerlegen. Dazu führt uns die nächste Folgerung:

Folgerung 15: Es gilt:

$$\overline{w}_z(h,s) := 1\text{-}w_z(h, s), \quad \overline{w}_z(h,1) = \frac{|S_{T(z)}| - h}{|S_{T(z)}|}, \quad \overline{w}_z(h,s+1) = \overline{w}_z(h,s) \cdot \frac{|S_{T(z)}| - h - s}{|S_{T(z)}| - s} \qquad (2.8.3)$$

Beweis: Siehe Anhang C ■

Natürlich gilt $\overline{w}_z(h,0) = 1$, denn wenn die Schmuggelumgebung leer ist, ist die Schnittmenge zwischen Schmuggelumgebung und Werteumgebung des betrachteten Werts (unabhängig von seinem Histogrammwert) auch leer und der Wert wurde damit nicht geschmuggelt, d. h. die Wahrscheinlichkeit, dass der betrachtete Wert geschmuggelt wurde erst gleich null und die Wahrscheinlichkeit, dass der betrachtete Wert nicht geschmuggelt wurde ist damit gleich eins.

Die Definition von $\overline{w}_z(h,1)$ ist konsistent, denn es gilt:

$$\overline{w}_z(h,1) = \overline{w}_z(h,0) \cdot \frac{|S_{T(z)}| - h - 0}{|S_{T(z)}| - 0} = 1 \cdot \frac{|S_{T(z)}| - h}{|S_{T(z)}|} = \frac{|S_{T(z)}| - h}{|S_{T(z)}|}. \qquad (2.8.4)$$

Für ganze Zahlen (insbesondere die Mächtigkeit der Schmuggelumgebung) ist die Schmuggelwahrscheinlichkeit nun mit dem Rechner rekursiv einfach berechenbar. Allerdings rechnen wir in den meisten Fällen mit abgeschätzten Durchschnittswerten oder Erwartungswerten, so dass sich als abgeschätzte Mächtigkeit der Schmuggelumgebung $s_{s|z|\tau}$ eine reelle Zahl ergibt.

Um diesem Problem zu begegnen, interpolieren wir die Schmuggelwahrscheinlichkeit für reelle s (da Histogramme Häufigkeiten als ganze Zahlen liefern, müssen wir nicht für h interpolieren) zwischen seinen ganzzahligen Nachbarn.

Folgerung 16: Sei $s \geq 0$ rational und

$$\tilde{w}_z(h,s) = \overline{w}_z(h,\lfloor s \rfloor) \cdot \left(\frac{|S_{T(z)}| - h - \lceil s \rceil}{|S_{T(z)}| - \lceil s \rceil} \right)^{s-\lfloor s \rfloor}. \qquad (2.8.5)$$

Dann gilt: $\tilde{w}_z(h,s) = \overline{w}_z(h,s)$ für ganzzahlige s und $\overline{w}_z(h,\lfloor s \rfloor) \geq \tilde{w}_z(h,s) \geq \overline{w}_z(h,\lceil s \rceil)$.

Beweis: Siehe Anhang C ■

Wir können nun $\tilde{w}_z(h,s)$ mit dem Computer berechnen und erhalten damit ein sinnvolles Ergebnis für $w_z(h,s)$, auch für reelle Parameter.

```
float w(float h, float s, Spalte z)
{
      // Initialisiere Ergebnis-Variable w mit 1
      float w = 1;
      // bilde Produkt mit Hilfe einer Schleife iterativ
      for int i=0 to floor(s) {
            w *= (|S_T(z)|-h-i)/(|S_T(z)|-i);
      }
      // interpoliere gemäß dem Fließkomma-Anteil von n und
      return 1 - w * pow((|S_T(z)|-h-ceil(s))/(|S_T(z)|-ceil(s)), s-floor(s));
}
```

Abbildung 9: Algorithmus zur Berechnung $w_b(h_b(\varphi))$.

Abbildung 9 zeigt einen Algorithmus zur iterativ Berechnung von $w_z(h, s)$, wobei h der Histogrammwert eines beliebigen Werts in der DBD des Schmuggelziels z und s die (abgeschätzte) Mächtigkeit der Schmuggelumgebung in z ist.

Abschließend wollen wir noch die Anzahl der in das Schmuggelziel eingeführten Werte berechnen. Diese Werte kommen beim Nachfolger des Schmuggelziels an und werden dort induziert, sofern sie in der DBD des Nachfolgers liegen.

Die Anzahl der geschmuggelten Werte hängt natürlich ebenfalls von der Mächtigkeit der Schmuggelumgebung ab.

Ist die Spalte unique, d. h. wenn die Kardinalität der Spalte gleich der Kardinalität der Tabelle ist, so ist die Frage nach der Anzahl der geschmuggelten Werte einfach zu beantworten: in diesen s Zeilen befinden sich genau s Werte, da die betrachteten Werte in verschiedenen Sätzen ebenfalls verschieden sind.

Ist dieses Kriterium aber nicht erfüllt, ist die Frage schwieriger zu beantworten. Wir können dann nur aufbauend auf die Schmuggelwahrscheinlichkeit die Anzahl der geschmuggelten Werte abschätzen. Die nächste Folgerung zeigt diesen Zusammenhang.

Folgerung 17: Sei z ein Schmuggelziel und s die abgeschätzte Mächtigkeit der Schmuggelumgebung in z. Dann gilt:

$$v_{s|z} = \sum_{\varphi \in V_z} w_z(h_z(\varphi), s) . \qquad (2.8.6)$$

Beweis: Siehe Anhang C ■

Die Berechnung dieser Größe ist natürlich sehr aufwendig. Zur Vereinfachung gehen wir nun von der Gleichverteilungsannahme aus. Unter dieser Annahme gilt

$$h_z(\varphi) = |S_{T(z)}| / c_z \text{ für alle } \varphi \in V_z .$$

Dann sagt uns die nächste Folgerung, wie man die Anzahl der in z geschmuggelten Werte abschätzen kann.

In [HB04a] wird eine Berechnungsvorschrift zur Abschätzung der Anzahl in eine Spalte geschmuggelten Werte genannt, die implizit ebenfalls auf die Schmuggelwahrscheinlichkeit unter den hier ebenfalls verwenden Annahmen (Gleichverteilungsannahme, Spaltenunabhängigkeitsannahme, Abstraktion von Null-Werten) aufbaut:

Folgerung 18: Sei z ein Schmuggelziel und s die abgeschätzte Anzahl der Sätze, die zum Schmuggeln von Werten in z beiträgt. Des Weiteren gelte die Gleichverteilungsannahme. Dann kann $v_{s|z}$ folgendermaßen berechnet werden:

$$v_{s|z} = c_z \cdot w_z(\left|S_{T(z)}\right| / c_z, s) =: c_z \cdot (1 - g_z(s)) =: f_z(s). \qquad (2.8.7)$$

Beweis: Siehe Anhang C ■

2.10 Zusammenfassung

In Kapitel 2 haben wir allgemeine Beobachtungen zu Cache-Groups und -Föderationen vorgestellt. Wir benötigen diese Beobachtungen und Zusammenhänge in den folgenden Abschnitten zur Modellierung des Füllverhaltens von Cache-Groups bzw. Cache-Föderationen.

In den Algorithmen zur quantitativen Analyse des Füllverhaltens in einer Cache-Group oder -Föderation verwenden wir immer wieder Annahmen zur Vereinfachung der Berechnungen. Wir haben zunächst diese Annahmen definiert und unter die Lupe genommen. Die Gleichverteilungsannahme besagt, dass die Werte in den Spalten im Backend über den Wertebereich gleichverteilt sind. Die Spaltenunabhängigkeitsannahme besagt, dass die Verteilung der Werte in verschiedenen Spalten stochastisch unabhängig ist. Auch die Betrachtung von Null-Werten würde die Modellierung enorm erschweren. Aus diesem Grund gehen wir grundsätzlich davon aus, dass keine Null-Werte in den Spalten vorkommen.

Der kennen gelernte Spalten-Beziehungsgraph repräsentiert die wichtigsten Elemente einer Cache-Group, d. h. die Spalten sowie Zusammenhänge zwischen diesen. Im Gegensatz zu einem einfachen Erreichbarkeitsgraph, der lediglich Cache-Tabellen durch Knoten und RCCs durch dazwischen liegende gerichtete Kanten darstellt, erhalten wir mit dem SBG auch direkt Informationen über die Beziehungen der Spalten innerhalb einer Cache-Tabelle. Dieses Mehr an Information wird von den Algorithmen zur Berechnung der Füllstände einer Cache-Group oder -Föderation aufgegriffen. Mit Hilfe des SBG können wir die Spalten kategorisieren: Eine

Spalte heißt Konsument, wenn an ihr eine RCC-Kante endet und Erzeuger, wenn an ihr eine RCC-Kante anfängt. Wir bezeichnen eine Spalte, die ein Erzeuger ist, aber kein Konsument, als Basiserzeuger. Eine Spalte heißt Schmuggelquelle, wenn an ihr eine Schmuggler-Kante beginnt und Schmuggelziel, wenn an ihr eine Schmuggler-Kante endet.

Des Weiteren haben wir verschiedene Zusammenhänge zwischen Wertemengen kennen gelernt. Während eines Füllschritts gelangen neue Werte in die verschiedenen Spalten einer Cache-Tabelle. Ein Wert φ heißt zum Zeitpunkt τ in Spalte c zusätzlich geladen (bzw. zusätzlich umgebungsvollständig, zusätzlich induziert, zusätzlich geschmuggelt), wenn φ vor τ noch nicht in c geladen (bzw. umgebungsvollständig, induziert, geschmuggelt) war, es aber unmittelbar danach ist. Wir haben im einzelnen die DBD einer Spalte partitioniert und die Aufteilung der geladenen Werte in induzierte und geschmuggelte Werte betrachtet.

Da bei der Modellierung des Füllverhaltens von Cache-Groups und -Föderationen oft die Mächtigkeit einer Schnittmenge berechnet werden muss, haben wir im nächsten Abschnitt in einem Exkurs Betrachtungen zur Abschätzung der Schnittmengenmächtigkeit gemacht. Hierbei haben wir die Information benutzt, dass wir zu zwei Mengen A und B annehmen, dass sie unabhängig in einer gemeinsamen Obermenge C liegen. Die Mächtigkeit der Schnittmenge wurde dann durch die Formel

$$|A \cap B| \approx |A| \cdot |B| / |C|$$

abgeschätzt.

Im Rahmen dieser Arbeit werden Histogramme – welche wir ebenfalls betrachtet haben – zur Verbesserung der Abschätzung des Füllverhaltens bei baumartigen Cache-Groups, also in Kapitel 3, eingesetzt. Bei allgemeinen, insbesondere zyklenbehafteten Cache-Groups würde die Modellierung hierdurch so komplex, dass die Verbesserung der Abschätzung die Komplexität der Modellierung nicht rechtfertigen würde.

Die Zugehörigkeitsfunktion sagt uns, ob ein Wert überhaupt in der DBD einer Spalte liegt. Die kombinierte Zugehörigkeitsfunktion erweitert die Funktion auf die Schnittmenge mehrerer DBDs. Natürlich können beide Funktionen nur mit der Kenntnis der konkreten DBD einer Spalte (bzw. mehrerer Spalten) berechnet werden. Über die DBD einer Spalte geben uns aber wiederum Histogramme Auskunft.

Wir können für die Beantwortung der Frage nach der Anzahl der beim ersten Füllschritt in die Wurzel-Tabelle geladenen Sätze zwei Situationen unterscheiden. Zum einen ist die Wahrscheinlichkeit für jeden Cache-Key-Wert gleich hoch, dass er innerhalb einer festen Zeitspanne referenziert wird. In diesem Fall ergibt sich die durchschnittliche, erwartete Werteumgebung der möglichen Cache-Key-Werte, indem man die Kardinalität der Backend-Tabelle durch die Anzahl der Werte in der DBD des Cache-Keys teilt. Zum anderen ist die Referenzierungswahrscheinlichkeit explizit bekannt. Hier berechnet sind der Erwartungswert für die Mächtigkeit der Werteumgebung des referenzierten Cache-Key-Werts als Summe von Produkten aus Häufigkeit und Referenzierungswahrscheinlichkeit für jeden Wert aus der DBD des Cache-Keys.

RCCs und Wertemengen-Pfade

RCCs haben ebenfalls großen Einfluss auf das Füllverhalten einer Cache-Group oder -Föderation, da die Werteumgebung der über eine RCC induzierten Werte direkt den Füllstand einer Cache-Tabelle beeinflusst. Dementsprechend haben wir in diesem Abschnitt RCCs und ihren Einfluss auf das Füllverhalten genauer betrachtet. Die RCC-Implikation ergibt sich direkt aus der RCC-Eigenschaft. Nach der RCC-Implikation ist ein Wert nach dem Füllzyklus induziert, wenn er im Vorgänger geladen und in der DBD vorhanden ist:

$$\varphi \in V_{l|e} \wedge \chi_k(\varphi) = 1 \Rightarrow \varphi \in V_{i|k}.$$

Im Allgemeinen gilt die RCC-Implikation tatsächlich nur in der gezeigten Richtung. Nur in Sonderfällen kann die RCC-Implikation auch für die andere Richtung gezeigt werden. Dies gilt unter anderem für isolierte RCCs, bei deren Konsumenten lediglich diese eine RCC endet, wobei dieser Konsument der einzige in seiner Cache-Tabelle ist. Die RCC-Implikation hat des Weiteren positive Nebenwirkungen für die Sondierung des Caches.

Eine große Rolle für die Modellierung dieses Verhaltens einer RCC spielt die Schnittmenge der DBDs von Erzeuger und Konsument. Je größer die Schnittmenge und damit die Überdeckung ist, desto mehr Werte werden im Durchschnitt über die betrachtete RCC induziert. Haben wir keine Informationen über die Mächtigkeit der Schnittmenge der DBDs von Erzeuger und Konsument, so arbeiten wir mit der RCC-Teilmengen-Annahme. Diese besagt, dass dann die kleinere der beiden DBDs eine Teilmenge der größeren DBD ist.

Wir haben für die Anzahl $v_{i|e\to k}$ der über eine RCC $e\to k$ induzierten Werte folgende Schranken gefunden:

$$\max(0,\ v_{l|e} + \ddot{u}_{k,e}\cdot c_e - c_e) \le v_{i|e\to k} \le \min(v_{l|e},\ \ddot{u}_{k,e}\cdot c_e)$$

Unter der Annahme, dass die Wahrscheinlichkeit für jeden Wert gleich hoch ist, in e geladen zu sein, können wir im Sinne einer Average-Case-Analyse die Anzahl der über die RCC induzierten Werte wie folgt annähern:

$$v_{i|e\to k} = v_{l|e}\cdot\ddot{u}_{k,e}$$

Ein zusammenhängender, zyklenfreier Weg von RCC-Kanten in einem SBG heißt Wertemengen-Pfad (WMP). Solch ein WMP erweitert gewissermaßen die RCC-Implikation für einzelne Werte transitiv: Ein Wert, der in den DBDs aller Spalten entlang alles WMP vorhanden und in der Spalte am Anfang des WMP geladen ist, wird in die Spalten auf dem WMP induziert.

Im Allgemeinen hat ein Konsument mehr als einen Basiserzeuger, der mit ihm über einen WMP verbunden ist. Nur bei isolierten WMPen, bei denen jede beteiligte RCC isoliert ist, ist zu jedem Konsumenten ein einziger Basiserzeuger eindeutig bestimmt und kann algorithmisch gefunden werden. Isolierte WMP haben weiterhin einen Einfluss auf die über sie induzierten Werte, was eine Modellierung sehr vereinfacht: Die RCC-Implikation gilt für alle RCCs in beide Richtungen, die Mengen der indizierten Werte in den Konsumenten auf dem WMP schließen sich ein und die Mächtigkeiten nehmen zumindest nicht zu, fallen also monoton entlang den Spalten auf dem WMP vom Basiserzeuger zum letzten Konsumenten.

Des Weiteren müssen alle in die Konsumenten induzierten Werte aus dem gleichen Basiserzeuger stammen, d. h. sie müssen dort geladen sein. Wir können dementsprechend die Wahr-

scheinlichkeit, dass ein Wert φ in einen Konsumenten k auf dem WMP induziert wird, berechnen. Diese Ladewahrscheinlichkeit ist sowohl abhängig von der kombinierten Zugehörigkeitsfunktion aller Spalten zwischen k und seinem Basiserzeuger b für φ, als auch von der Ladewahrscheinlichkeit von φ in b. Die Ladewahrscheinlichkeit eines Werts φ im Basiserzeuger b hängt von seiner Schmuggelwahrscheinlichkeit und Referenzierungswahrscheinlichkeit ab.

Mit Hilfe der Ladewahrscheinlichkeit kann der Füllstand der Cache-Tabellen z. B. in baumartigen Cache-Groups (aber auch allgemein in Cache-Tabellen entlang eines isolierten WMP) mit Hilfe von Histogrammen abgeschätzt werden, da jeder Konsument am Ende eines isolierten WMPs liegt.

Schmuggler-Beziehungen

Neben RCCs haben Schmuggler-Beziehungen ebenfalls einen großen Einfluss auf das Füllverhalten von Cache-Groups. Auf der Ebene einzelner Werte ist die Schmuggelwahrscheinlichkeit an die Ladewahrscheinlichkeit in den Konsumenten auf den abgehenden isolierten WMPen eines Basiserzeugers gekoppelt. Auf der Ebene der Menge der in ein Schmuggelziel geschmuggelten Werte steigt mit größer werdender Anzahl auch die Anzahl der „wandernden“ Werte, die an nachfolgenden Spalten ankommen. Diese erhöhte Anzahl führt dann wiederum zu größeren Füllständen in den Nachfolger-Tabellen.

Wir haben aus diesem Grund die Schmuggelwahrscheinlichkeit betrachtet und uns angeschaut, wie man diese effizient mit dem Rechner bestimmen kann. Hierzu haben wir zwei Möglichkeiten vorgestellt: Zum einen haben wir die Schmuggelwahrscheinlichkeit mit Hilfe einer Stirling-Formel angenähert. Zum anderen haben wir eine rekursive Form der Schmuggelwahrscheinlichkeit hergeleitet.

3 Baumartige Cache-Groups

In diesem Abschnitt betrachten wir das Füllverhalten einer bestimmten Klasse von Cache-Groups, deren SBG ein Baum ist. Insbesondere erfüllen die betrachteten Cache-Groups die Voraussetzung, dass bei jeder Cache-Tabelle höchstens eine RCC endet. Als weitere Voraussetzung nehmen wir an, dass es in der Cache-Group nur einen Cache-Key gibt.

Definition: Wir nennen eine Cache-Group **baumartig**, wenn der zugehörige SBG ein Baum ist. Wir nennen den SBG in diesem Fall auch Spalten-Beziehungsbaum (SBB).

Der SBG ist genau dann ein Baum, wenn er keine Zyklen enthält und jeder Knoten maximal einen Vater hat. Jede Spalte außer dem Cache-Key hat dabei mindestens einen Vater.

Abbildung 10 zeigt auf der linken Seite eine baumartige Cache-Group als Erreichbarkeitsgraph. Diese Cache-Group ist aus der in Abbildung 4 gezeigten Cache-Group durch Weglassen der beiden RCCs B.acl→A.acl und D.did→E.did entstanden. Auf der rechten Seite dieser Abbildung ist die gleiche Cache-Group als SBG zu sehen. Der SBG ist ein Baum und deswegen ist die Cache-Group baumartig.

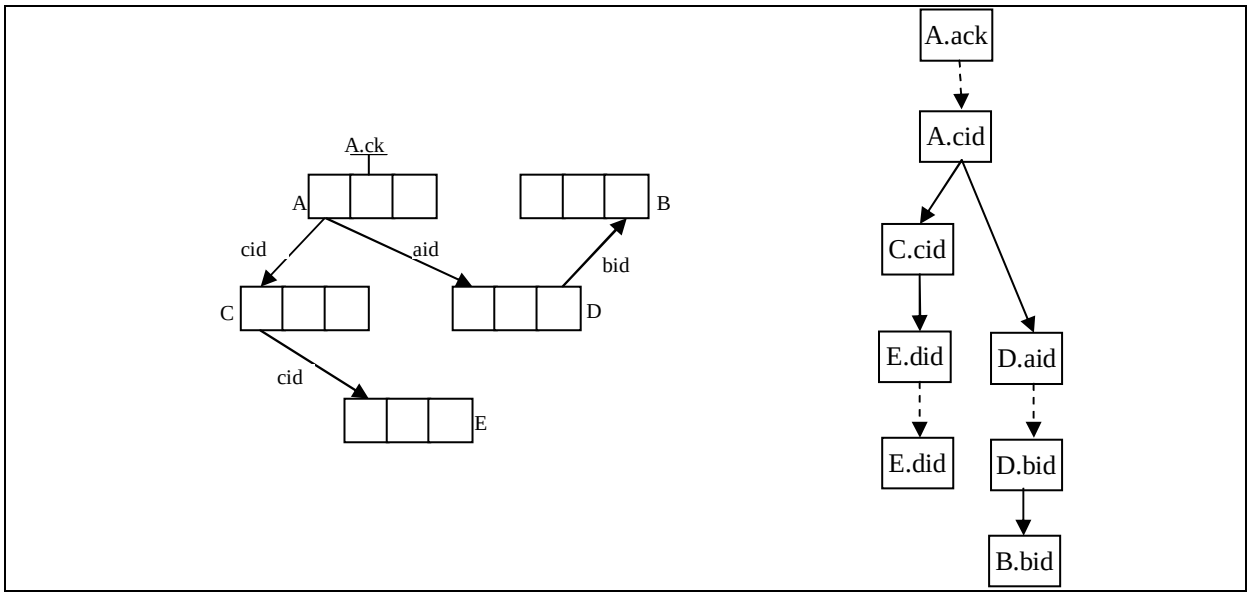

Abbildung 10: Eine baumartige Cache-Group als Erreichbarkeitsgraph und als SBB

Im Folgenden untersuchen wir zunächst, welche direkten Schlussfolgerungen sich aus der Baumartigkeit ergeben und welche grundlegenden Eigenschaften die betrachteten Cache-Groups besitzen. Anschließend betrachten wir das Laden von Sätzen in die Cache-Tabellen. Wir untersuchen dazu zunächst das Füllverhalten der Wurzel-Tabelle (d. h. der Cache-Tabelle, welche den Cache-Key enthält).

Danach untersuchen wir das Füllverhalten der Cache-Tabellen weiter unten im Erreichbarkeitsgraph.

Allgemein sagen wir, dass eine Cache-Tabelle T_1 **weiter unten** oder **unterhalb** im Vergleich zu einer anderen Cache-Tabelle T_2 in einer baumartigen Cache-Group ist, wenn ein Weg mit mindestens einer RCC-Kante von einer Spalte aus T_1 zu einer Spalte aus T_2 existiert.

Anschließend entwickeln wir in Abschnitt 3.4 mehrere Algorithmen auf Basis des in den vorangegangenen Abschnitten gewonnenen Wissens, um den Füllstand aller Cache-Tabellen einer baumartigen Cache-Group abzuschätzen.

3.1 Grundlegende Zusammenhänge und Eigenschaften

In diesem Abschnitt wollen wir grundsätzliche Zusammenhänge und Eigenschaften baumartiger Cache-Groups betrachten.

Durch die baumartige Struktur der betrachteten Cache-Group ergibt sich vor allem die Eindeutigkeit der Konsumenten, eingehenden RCCs und Basiserzeuger jeder Cache-Tabelle.

Folgerung 19: Sei T eine Cache-Tabelle einer baumartigen Cache-Group. Dann existiert in T maximal ein Konsument. Ist T die Wurzel-Tabelle der Cache-Group (d. h. beinhaltet T den Cache-Key), so hat T keinen Konsumenten.

Beweis: Sei zunächst T die Wurzel-Tabelle. Dann kann T keine eingehende RCC haben. Ansonsten wäre diese RCC ein Teilweg am Ende eines nicht-leeren Wegs[19] zwischen Cache-Key und T. Dies wäre aber ein Widerspruch zur Annahme, dass T die Wurzel-Tabelle ist. Sei nun T nicht die Wurzel-Tabelle. Dann endet an T genau eine RCC. Diese RCC existiert und ist der Teilweg am Ende des nicht-leeren Wegs zwischen Cache-Key und T: Ansonsten wäre T nicht im Erreichbarkeitsgraph der Cache-Group und somit keine Spalte von T im SBB enthalten. Die eindeutig identifizierte RCC endet an einem Konsumenten von T. Dies ist der einzige Konsument, da sonst keine weitere RCC bei T enden darf. Enden mehr als eine RCC bei T, so wäre der SBG der Cache-Group nicht baumartig. ■

Bemerkung: Diese Folgerung besagt implizit auch, dass alle WMPe in einer baumartigen Cache-Group isoliert sind. Dadurch können wir alle Eigenschaften nutzen, die isolierte WMPe mit sich bringen, um die Abschätzung der Füllstände in der Cache-Group zu verbessern. Denn an jeder Cache-Tabelle endet nur eine RCC, dementsprechend ist jede RCC isoliert und damit auch jeder WMP.

Schreibweise: Wir schreiben für den Konsumenten $k_T \in T$, wenn T die zugehörige Cache-Tabelle einer baumartigen Cache-Group ist. Falls eindeutig aus dem Kontext hervorgeht, zu welcher Cache-Tabelle der Konsument gehört, verzichten wir auf den Index.

[19] Ein nicht-leerer Weg ist ein Weg mit Länge größer 0.

Die Spalten einer Cache-Tabelle bilden zusammen mit den dazwischen liegenden Schmuggler-Kanten Teilbäume des gesamten SBB. Dabei wird der Konsument mit jedem Erzeuger durch eine Schmuggler-Kante verbunden.

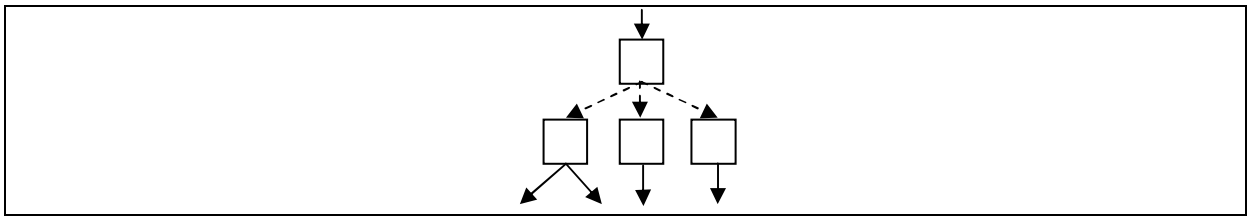

Abbildung 11: Ausschnitt aus einer baumartigen Cache-Group

In Abbildung 11 ist ein Ausschnitt einer baumartigen Cache-Group gezeigt. Man sieht eine Cache-Tabelle, deren Konsumenten mit eingehender RCC und drei Schmuggler-Kanten zu den Erzeugern der Cache-Tabelle. Diese haben wiederum eine oder mehrere ausgehende RCCs.

Dadurch, dass an jeder Cache-Tabelle nur eine RCC endet, existieren keine Schmuggler-Beziehungen, die am Konsumenten einer Cache-Tabelle enden. Deswegen können in einen Konsumenten niemals Werte geschmuggelt werden. Der einzige Weg, wie ein Wert in einen Konsumenten gelangen kann, ist, dass er induziert wird. Wir können somit den folgenden Zusammenhang festhalten:

Folgerung 20: Sei k ein Konsument in einer baumartigen Cache-Group. Dann gilt zu jedem Zeitpunkt τ:

$$V_{l|k|\tau} = V_{i|k|\tau} = V_{vc|k|\tau}.$$

Beweis: Wie wir wissen, existiert keine Schmuggler-Beziehung, die bei k endet. Dementsprechend ist die Menge der in k geschmuggelten Werte zu jedem Zeitpunkt leer. Aus Folgerung 1 (Abschnitt 2.4) ergibt sich:

$$V_{l|k|\tau} = V_{s|k|\tau} \cup V_{i|k|\tau} = \varnothing \cup V_{i|k|\tau} = V_{i|k|\tau}. \blacksquare$$

Eine ähnliche Folgerung ergibt sich auch für Basiserzeuger. Außer beim Cache-Key gelangen Werte in einen Basiserzeuger ausschließlich durch Schmuggeln. Hier kann also die umgekehrte Folgerung aufgestellt werden.

Folgerung 21: Sei b ein Basiserzeuger einer baumartigen Cache-Group, aber nicht der Cache-Key. Dann gilt zu jedem Zeitpunkt τ:

$$V_{l|b|\tau} = V_{s|b|\tau}.$$

Beweis: Nach den gegebenen Voraussetzungen ist der einzige Weg, über den Werte in eine solche Spalte gelangen können, dass sie in den Basiserzeuger geschmuggelt werden. Analog zu Folgerung 20 ergibt sich:

$$V_{l|k|\tau} = V_{i|k|\tau} \cup V_{i|k|\tau} = V_{i|k|\tau} \cup \varnothing = V_{i|k|\tau}.$$

Bemerkung: Folgerung 20 und Folgerung 21 gelten insbesondere nach dem Füllzyklus. In unseren weiteren Untersuchungen lassen wir dementsprechend gegebenenfalls den Suffix des Index zur Angabe des betrachteten Zeitpunkts weg.

3.2 Modellierung eines Füllzyklus

Nachdem wir nun die notwendigen Hilfsmittel zur Verfügung haben, werden wir einen ersten Ansatz machen, um das Füllverhalten einer baumartigen Cache-Group zu bewerten. Dazu betrachten wir in diesem Abschnitt einen einzelnen Füllzyklus.

Ein Füllzyklus beginnt mit dem ersten Füllschritt der Wurzel-Tabelle und endet, wenn die Cache-Group stabil ist. Während eines Füllzyklus werden eine Reihe von Füllschritten von den einzelnen Cache-Tabellen gemacht, um die RCC-Eigenschaft zu erfüllen.

Dadurch, dass die betrachteten Cache-Groups baumartig sind, wird insbesondere sichergestellt, dass die Cache-Groups keine Zyklen enthalten und dass es somit nicht zu einem rekursiven Laden kommen kann. Baumartige Cache-Groups sind also immer sicher. Dadurch sind die betrachteten Cache-Groups sehr einfach zu handhaben, und ihr Füllverhalten ist einfach zu berechnen.

Der erste Füllschritt wird in der Wurzel-Tabelle ausgeführt, um die Werteumgebung des durch eine Benutzeranfrage referenzierten Cache-Key-Werts in die Cache-Tabelle zu laden. Dadurch werden Nachfolger der Erzeuger ungesättigt, so dass ihre Cache-Tabelle ebenfalls einen Füllschritt ausführen muss.

Dieser Vorgang setzt sich fort bis zu den Blättern des SBB. Hier haben die Cache-Tabellen keine ausgehenden RCCs, so dass auch kein weiteres Laden in eventuellen Nachfolgern ausgelöst wird.

Auf Grund der baumartigen Struktur der betrachteten Cache-Groups führt jede Cache-Tabelle jedoch lediglich einen RCC-Füllschritt aus, was die Modellierung erheblich vereinfacht.

Wie wir gesehen haben, müssen wir bei der Betrachtung eines Füllschritts allgemein zwei Situationen unterscheiden. Zum einen ist es möglich, dass ein Füllschritt ausgeführt wird, weil ein Cache-Key-Wert durch eine Benutzeranfrage referenziert wurde. Dieser Füllschritt würde bei unseren betrachteten Cache-Groups von der Wurzel-Tabelle ausgeführt werden. In Abschnitt 2.7 haben wir gesehen, wie man die durchschnittliche Mächtigkeit der Werteumgebung des referenziert Cache-Key-Werts bestimmen kann. Dadurch, dass die Wurzel-Tabelle nur einen Füllschritt macht, können wir somit ihren Füllstand bestimmen.

Auf der anderen Seite wird auch dann ein Füllschritt ausgeführt, wenn eine Anzahl von Werten über eine RCC vom Vorgänger einer Cache-Tabelle das Laden von Werten verursacht. Einen solchen Füllschritt untersuchen wir in Abschnitt 3.3.

3.3 Verhalten von Cache-Tabellen ohne Cache-Key

In diesem Abschnitt wollen wir betrachten, wie die Füllschritte in den Cache-Tabellen ohne Cache-Key ausgeführt werden. Bei den nun betrachteten Cache-Tabellen handelt es sich also nicht um die Wurzel-Tabelle.

Wie wir gesehen haben, wird ein Füllschritt einer Cache-Tabelle T unterhalb der Wurzel-Tabelle der Cache-Group ausgelöst, weil der Konsument k_T ungesättigt ist. Es kommen also einige Werte über die eingehende RCC bei k_T an. Diese Werte müssen durch Nachladen von Sätzen aus dem Backend einstiegsfähig nach dem Füllschritt sein, damit die RCC-Eigenschaft erfüllt ist. Ein ankommender Wert ist also nach dem Füllschritt entweder induziert und damit umgebungsvollständig oder er ist nicht im Backend von k_T vorhanden.

Um die RCC-Eigenschaft zu erfüllen, wird während des Füllschritts die Werteumgebung der durch den Vorgänger des Konsumenten induzierten Werte in die Cache-Tabelle T geladen.

Die geladenen Sätze verursachen eventuell über Schmuggler-Beziehungen ein Schmuggeln von Werten in die Erzeuger der Cache-Tabelle. Diese geschmuggelten Werte bewirken dann, dass die Nachfolger des Erzeugers ungesättigt werden.

Eine Schmuggler-Beziehung in einer baumartigen Cache-Group wirkt immer auf einen Basiserzeuger.[20] Diese Spalte ist dabei nicht der Cache-Key.

Daraus, dass alle RCCs in einer baumartigen Cache-Group isoliert sind, ergibt sich, dass alle in einen Basiserzeuger geladenen Werte in ihn geschmuggelt wurden, wenn er ein Schmuggelziel ist.

Folgerung 22: Sei b ein Basiserzeuger und Schmuggelziel in einer baumartigen Cache-Group. Dann gilt:

$$V_{l|b} = V_{s|b}.$$

Beweis: Da b ein Basiserzeuger ist, endet bei ihm keine RCC. Der Basiserzeuger darf außerdem in diesem Fall nicht der Cache-Key sein. Aus diesem Grund können Werte nur durch Schmuggeln in b gelangen. ■

Da alle geladenen Werte geschmuggelt wurden, können wir das in Abschnitt 2.8 gewonnene Wissen verwenden, um die Anzahl der geschmuggelten und damit geladenen Werte zu bestimmen. Außerdem können wir ebenfalls die Schmuggelwahrscheinlichkeit für einen gegebenen Wert berechnen.

Weil jede Cache-Tabelle lediglich einen Füllschritt ausführt, müssen wir auch nicht zwischen temporärer und zusammenfassender bzw. endgültiger Schmuggelumgebung unterscheiden. Außerdem entspricht die Schmuggelumgebung der Menge der in die Cache-Tabelle geladenen Sätze. Dementsprechend gilt insbesondere die nächste Folgerung.

[20] Bei allgemeinen Cache-Groups und -Föderationen können sich Schmuggler-Beziehungen auch auf „normale“ Erzeuger (die keine Basiserzeuger sind) und sogar auf Spalten, die lediglich Konsumenten aber überhaupt keine Erzeuger sind auswirken.

Folgerung 23: Sei b ein Basiserzeuger in einer baumartigen Cache-Group, wobei T(b) nicht die Wurzel-Tabelle sei. Dann gilt:

$$\tilde{s}_{s|b} = n_{T(b)}.$$

Beweis: Zunächst wissen wir, dass b niemals die aktive Spalte sein kann. Dadurch tragen alle geladenen Sätze zum Schmuggeln von Werten in b bei. Die Menge der geladenen Sätze entspricht somit der Schmuggelumgebung von b und insbesondere sind auch die Mächtigkeiten gleich. ■

Unter der Gleichverteilungsannahme können wir nun mit Hilfe von Folgerung 18 die Anzahl der in einen Basiserzeuger geschmuggelten d. h. geladenen Werte abschätzen.

Folgerung 24: Sei b ein Basiserzeuger in einer baumartigen Cache-Group. Dann gilt unter der Gleichverteilungsannahme:

$$v_{l|b} = v_{s|b} = f_e(n_{T(b)}).$$

Beweis: Die Formel ergibt sich durch Anwendung der Folgerung 22*Folgerung 22* und Folgerung 23 auf Folgerung 18. ■

3.4 Algorithmus zur Abschätzung der Füllstände

In diesem Abschnitt wollen wir aufgrund des bisher gewonnen Wissens über baumartige Cache-Groups einen Algorithmus entwickeln, um die Anzahl der geladenen Sätze in den Cache-Tabellen der Cache-Group abzuschätzen. Dies geschieht zunächst unter der Gleichverteilungsannahme, der Spaltenunabhängigkeitsannahme und der RCC-Teilmengen-Annahme. Wir rekapitulieren noch einmal die Zusammenhänge und führen die bisher entwickelten Formeln auf. Anschließend entwickeln wir den gewünschten Algorithmus.

Diesen Algorithmus werden wir anhand eines Beispiels nachvollziehen. Danach und entwickeln einen Algorithmus unter Zuhilfenahme von Histogrammen.

3.4.1 Abschätzung der Füllstände ohne Histogramme

Betrachten wir Cache-Groups unter der Annahme der Gleichverteilungsannahme und fassen wir die bisher entwickelten Formeln zusammen.

Wie wir gesehen haben, wird ein Füllschritt ausgelöst, weil der Konsument ungesättigt ist. Die beim Konsumenten ankommenden Werte müssen durch Laden von Sätzen aus dem Backend einstiegsfähig gemacht werden, damit die RCC-Eigenschaft erfüllt ist. Ein ankommender Wert ist also nach dem Füllschritt entweder induziert und damit umgebungsvollständig oder er ist nicht im Backend von k_T vorhanden.

Da alle RCCs in einer baumartigen Cache-Group isoliert sind, können wir die Anzahl der induzierten Werte wie in Abschnitt 2.8.4 berechnen. Eine Average-Case-Analyse ohne Histogramme ergibt also, dass

$$v_{i|k} = v_{l|e} \cdot \ddot{u}_{k,e}$$

gilt, wenn wir den Konsumenten k einer Cache-Tabelle in einer baumartigen Cache-Group und seinen Vorgänger e betrachten.

Wir können die Häufigkeit von umgebungsvollständigen Werten im Rahmen einer Average-Case-Analyse ohne Histogramme berechnen, indem wir die Mächtigkeit der Wertemenge mit der durchschnittlichen Anzahl der Sätze pro Wert multiplizieren. In unserem Fall ergibt dies für die Menge der in den Konsumenten k induzierten Werte:

$$s_{i|k} = v_{i|k} \cdot |S_{T(k)}| \cdot / c_k.$$

Fasst man beide Formeln zusammen, so kann die Anzahl der in einer Cache-Tabelle T(k) geladenen Sätze in Zusammenhang mit der Anzahl der in den Vorgänger e des Konsumenten $k=k_{T(k)}$ geladenen Werte gebracht werden. Wir schreiben:

$$s_{i|k} = v_{l|e} \cdot \ddot{u}_{k,e} \cdot |S_{T(k)}| \cdot / c_k.$$

Im Wirkzusammenhang II betrachten wir, wie viele Werte ein Erzeuger enthält, nachdem eine gegebene Anzahl von Sätzen geladen wurde. Es gilt nach Folgerung 17:

$$v_{l|e} = f_e(n_{T(k)}).$$

Dabei ist $v_{l|e}$ die Anzahl der in den Erzeuger e geschmuggelten und somit in ihm geladenen Werte. f_e ist die fundamentale Funktion aus dem vorhergehenden Abschnitt.

In den Cache-Key ck wurde während des ersten Füllschritts auf Grund seiner Füllpunkteigenschaft ein einzelner Wert eingeführt, da wir den Zustand der Cache-Group nach einem einzelnen Füllzyklus betrachten und die Cache-Group vorher leer war:

$$v_{l|ck} = 1.$$

Mit diesem Wissen über die Anzahl der geladenen Werte im Konsumenten und den Erzeugern können wir nun den Zusammenhang verschiedener Cache-Tabelle-Konfigurationen im gesamten SBB betrachten.

Um das Füllverhalten einer solchen einfachen Cache-Group zu modellieren, können wir den folgenden einfachen Algorithmus verwenden. Als Eingabe erwartet der Algorithmus einen SBB mit zugehörigen statistischen Daten über die Kardinalitäten der Spalten und Cache-Tabellen.

```
void berechne_Füllstand(Spalte c, float v) {
      // berechne die Anzahl der zu ladenden Sätze in T(c).
      n_T(c) = v · |S_T(c)| / c_c;
      foreach (e ∈ E_T(c)) {
            foreach (l ∈ K_e) {
                  berechne_Füllstand(l, f_e(n_T(c)) · ü_c,e);
            }
      }
}
berechne_Füllstand(ck, 1);
```

Abbildung 12: Algorithmus zur Berechnung der Füllstände aller Cache-Tabellen einer baumartigen Cache-Group.

Abbildung 12 zeigt einen Algorithmus zur Berechnung der Füllstände aller Cache-Tabellen von baumartigen Cache-Groups.

In einer rekursiv definierten Prozedur wird zunächst der Füllstand der Cache-Tabelle einer Spalte c berechnet, wenn in sie v Werte geladen wurden. Hierzu erwartet die Funktion den Konsument c der Cache-Tabelle als Spalten-Objekt und die Anzahl der im Konsument geladenen Werte v als Fließkomma-Wert.

In einem ersten Schritt berechnen wir die Anzahl der in die Cache-Tabelle der Spalte c geladenen Sätze. Anschließend wird die Anzahl der in die verschiedenen Erzeuger geschmuggelten Werte berechnet. Für jeden dieser Erzeuger sucht die Prozedur nach verbundenen Konsumenten und berechnet durch Rekursion, wie viele Werte in diesen Konsumenten geladen werden. Der Algorithmus betrachtet also die Cache-Tabellen in topologischer Sortierordnung im Erreichbarkeitsgraph.

Mit diesem Algorithmus erhalten wir für jede beteiligte Cache-Tabelle in der Cache-Group einen Erwartungswert für die Anzahl der geladenen Zeilen aus dem Backend. Eine Verwendung von Histogrammen ist hierbei nicht notwendig.

Haben wir kein explizites Wissen über die Überdeckungen zwischen jeweils zwei mit einer RCC verbundenen Spalten – d. h. kennen wir die Größe $ü_{k,e}$ für Erzeuger e und Konsumenten k mit $e \rightarrow k$ nicht – so können die Überdeckung mit Hilfe der RCC-Teilmengen-Annahme abschätzen. In diesem Fall ersetzen wir die Berechnungsvorschrift zur Abschätzung der in den Konsumenten geladen Werte und schreiben:

$$v_{l|k} = v_{l|e} \cdot \min(1, c_k/c_e)$$

statt

$$v_{l|k} = v_{l|e} \cdot ü_{k,e.}$$

Man beachte, dass die konkrete Implementierung der Funktion f_e nicht vorgegeben ist. Man kann hier beliebig eine der oben beschriebenen Varianten verwenden.

3.4.2 Betrachtung des Algorithmus anhand eines Beispiels

Betrachten wir den Algorithmus anhand eines Beispiels. Abbildung 13 zeigt die modellierte Cache-Group als Erreichbarkeitsgraph links und als SBB rechts. In den Spalten des SBB ist die statistische Information über die Spalten dargestellt. Die Bezeichnung c_i zeigt dabei die Kardinalität der Spalte i. Zusätzlich sind in den Konsumenten die Kardinalitäten der entsprechenden Cache-Tabelle dargestellt, hier symbolisch als $\left|S_{T(c_i)}\right|$ für die zu Spalte i gehörende Cache-Tabelle geschrieben.

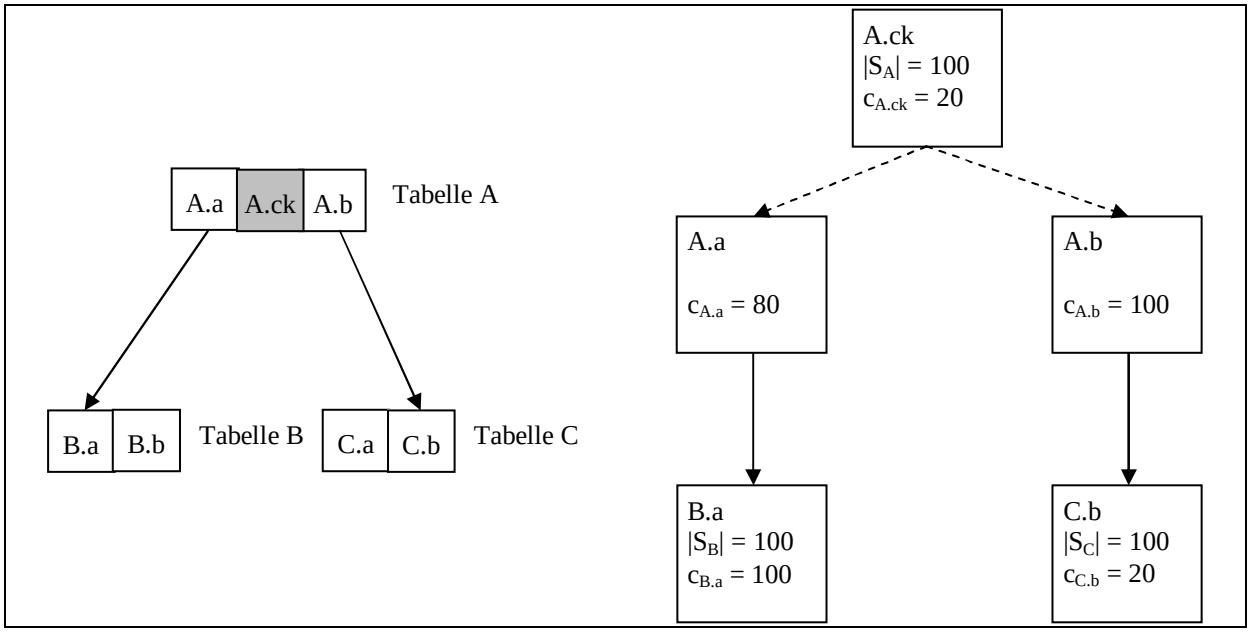

Abbildung 13: modellierte Cache-Group als Erreichbarkeitsgraph und als SBB

Die Cache-Group besteht aus drei Tabellen A, B und C. Tabelle A hat drei Spalten, A.a, A.ck und A.b. Der Cache-Key ist die Spalte A.ck. Dementsprechend ist die Spalte die Wurzel des SBB. Im Erreichbarkeitsgraph wird dies durch eine graue Schattierung deutlich gemacht. Die Tabellen B und C haben die Spalten B.a und B.b bzw. C.a und C.b. In der Cache-Group existieren zwei RCCs. Die erste RCC beginnt in der Wurzel-Tabelle bei Spalte A.a und endet bei B.a in Tabelle B. Die zweite RCC beginnt ebenfalls in der Wurzel-Tabelle, aber in Spalte A.b und endet bei Tabelle C in Spalte C.b.

Erzeuger der Wurzel-Tabelle sind die Spalten A.a und A.b. Dementsprechend existieren hier zwei Schmuggler-Beziehungen: Die erste verbindet den Cache-Key A.ck mit dem Erzeuger A.a, und die zweite verbindet den Cache-Key mit dem Erzeuger A.b. Die entsprechenden Schmuggler-Kanten sind durch einen gestrichelten Pfeil dargestellt.

Erzeuger A.a ist über eine RCC mit Konsument B.a verbunden. Dementsprechend existiert im SBB eine RCC-Kante von A.a nach B.a. Genauso existiert eine RCC-Kante von A.b nach C.b, da auch diese beiden Spalten durch eine RCC verbunden sind. RCC-Kanten sind in Abbildung 13 durch durchgezogene Kanten dargestellt.

Die Spalten B.b und C.a sind nicht im SBB vorhanden, da sie keine Erzeuger sind und das Füllverhalten der Cache-Group nicht beeinflussen.

Wir wollen nun betrachten, wie mit Hilfe des oben dargestellten Algorithmus das Füllverhalten der Cache-Group aus Abbildung 13 nachvollzogen werden kann und wir so die Füllstände der Cache-Tabellen abschätzen können.

Wir nehmen dazu an, dass die Funktion f rekursiv berechnet wird. Dies ist im Beispiel problemlos, insbesondere ohne Interpolation möglich, da wir in der einzigen Tabelle mit Schmuggler-Beziehungen – nämlich der Wurzel-Tabelle – lediglich einen ganzzahligen Wert für die Anzahl der geladenen Sätze einzusetzen haben. Es gilt also:

$$f_e(n_{T(e)}) = c_e \cdot (1 - g_e(n_{T(e)}) \qquad g_e(x+1) = g_e(x) \cdot \frac{|S_{T(e)}| - |S_{T(e)}|/c_e - x}{|S_{T(e)}| - x} \qquad g_e(0) = 1.$$

Dabei ist e der betrachtete Erzeuger und $n_{T(e)}$ die Anzahl der in seine Cache-Tabelle geladenen Sätze.

Zunächst wird die Anzahl der Sätze, die in die Wurzel-Tabelle geladen werden, berechnet:

$$n_A = 1 \cdot N_A / c_{A.ck} = 100 / 20 = 5.$$

Als nächstes wird jeder Erzeuger der Cache-Tabelle A betrachtet. Das sind die beiden Spalten A.a und A.b. Betrachten wir zunächst A.a.

Wir berechnen wie viele Werte in die betrachtete Spalte geschmuggelt wurden. Wir berechnen $g_{A.a}(n_A)$ iterativ:

$$g_{A.a}(0) = 1,$$

$$g_{A.a}(1) = g_{A.a}(0) \cdot \frac{|S_{A.a}| - |S_{A.a}|/c_e}{|S_{A.a}|} = 1 \cdot \frac{100 - 100/80}{100} = 0{,}9875,$$

$$g_{A.a}(2) = g_{A.a}(1) \cdot \frac{|S_{A.a}| - |S_{A.a}|/c_e - 1}{|S_{A.a}| - 1} = 0{,}9875 \cdot \frac{100 - 100/80 - 1}{100 - 1} = 0{,}9875 \cdot \frac{99 - 1{,}25}{99} = 0{,}975,$$

$$g_{A.a}(3) = g_{A.a}(2) \cdot \frac{|S_{A.a}| - |S_{A.a}|/c_e - 2}{|S_{A.a}| - 2} = 0{,}975 \cdot \frac{98 - 1{,}25}{98} = 0{,}9626,$$

$$g_{A.a}(4) = g_{A.a}(3) \cdot \frac{|S_{A.a}| - |S_{A.a}|/c_e - 3}{|S_{A.a}| - 3} = 0{,}9626 \cdot \frac{97 - 1{,}25}{97} = 0{,}9502,$$

$$g_{A.a}(n_A) = g_{A.a}(5) = g_{A.a}(4) \cdot \frac{|S_{A.a}| - |S_{A.a}|/c_e - 4}{|S_{A.a}| - 4} = 0{,}9502 \cdot \frac{96 - 1{,}25}{96} = 0{,}9378.$$

$$f_{A.a}(n_A) = c_{A.a} \cdot (1 - g_{A.a}(n_A)) = 80 \cdot (1 - 0{,}9378) = 80 \cdot 0{,}0622 = \mathbf{4{,}976}$$

In den Erzeuger A.a werden also durchschnittlich 4,976 Werte geschmuggelt.

Wir betrachten nun alle Konsumenten, die mit dem Erzeuger über eine RCC verbunden sind. In unserem Beispiel gibt es nur einen solchen Konsumenten: B.a.

Es wird im folgenden Schritt berechnet, wie viele Werte in den betrachteten Konsumenten geladen werden. Der betrachtete Konsument ist B.a. Da der Konsument eine größere Kardinalität hat als der Erzeuger und wir die RCC-Teilmengen-Annahme anwenden, werden alle in den Erzeuger geschmuggelten Werte auch in den Konsumenten geladen.

Wir berechnen nun rekursiv, wie viele Sätze in die Cache-Tabelle B geladen werden, unter der Voraussetzung, dass in den Konsumenten B.a 4,976 Werte geladen werden.

Wie zu Beginn des vorhergehenden Aufrufs auch berechnen wir die Anzahl der geladenen Sätze:

$$n_B = v_{l|B.a} \cdot N_B / c_{B.a} = 4{,}976 \cdot 100 / 100 = 4{,}976.$$

In die Cache-Tabelle B werden also 4,976 Sätze geladen. Dies ist auch logisch nachvollziehbar, wenn man bedenkt, dass der Konsument unique ist. In diesem Fall werden in die Cache-Tabelle immer genau so viele Sätze geladen, wie Werte in den Konsumenten geladen werden.

An dieser Stelle würde der Algorithmus fortfahren, indem er jeden Erzeuger der Cache-Tabelle betrachtet. Da hier allerdings keine Erzeuger existieren, wird der Schritt übersprungen und wir kehren zurück zum Aufrufer.

Hier wird der nächste Erzeuger der Cache-Tabelle A betrachtet. Es handelt sich dabei um die Spalte A.b. Wir berechnen wieder, wie viele Werte in den Erzeuger geschmuggelt werden. Wir wollen die Berechnungen an dieser Stelle nicht im Detail betrachten. Sie können analog zum vorhergehenden Erzeuger nachvollzogen werden.

Da die Kardinalität der Spalte A.b gleich der Kardinalität ihrer Cache-Tabelle A ist und der Erzeuger somit unique ist, können wir die Anzahl der in den Erzeuger geschmuggelten Werte aber auch berechnen, wenn wir uns Folgendes vor Augen führen: Jeder Satz referenziert einen Wert und alle referenzierten Werte sind unterschiedlich, da die Spalte unique ist. Dementsprechend ist die Anzahl der geschmuggelten Werte gleich der Anzahl der geladenen Sätze.

Wir berechnen, dass 5 Werte in den Erzeuger A.b geschmuggelt werden.

Im nächsten Schritt betrachten wir wieder die Konsumenten, die mit dem Erzeuger über eine RCC verbunden sind. In unserem Beispiel gibt es einen solchen Konsumenten: C.b.

Wir berechnen nun, wie viele Werte in C.b geladen werden. In diesem Fall ist die Kardinalität des Konsumenten kleiner als die Kardinalität des Erzeugers. Aus diesem Grund können nicht alle Werte aus der DBD des Erzeugers in der DBD des Konsumenten vorhanden sein. Wir wenden wiederum die RCC-Teilmengen-Annahme an und berechnen:

$$v_{l|C.b} = v_{l|A.b} \cdot \min(1, c_{C.b}/c_{A.b}) = 5 \cdot 20 / 100 = 1.$$

In den Konsumenten C.b wird also durchschnittlich ein Wert geladen.

Wie gehabt berechnen wir nun zunächst, wie viele Sätze in die Cache-Tabelle C geladen werden. Wir rechnen:

$$n_C = v_{l|C.b} \cdot N_C / c_{C.b} = 1 \cdot 100 / 20 = 5.$$

In die Cache-Tabelle C werden also durchschnittlich 5 Sätze geladen.

Der Algorithmus würde jetzt mögliche Erzeuger der Cache-Tabelle C betrachten. Da hier aber ebenfalls keine Erzeuger vorhanden sind, sind wir in diesem Zweig des Baums fertig. In der Cache-Tabelle A sind auch keine weiteren Erzeuger mehr vorhanden, die betrachtet werden müssen. Wir sind also fertig.

3.5 Algorithmus mit Histogrammen

Die Restriktionen bzw. Annahmen des Modells, das zum Algorithmus in Abschnitt 3.4.1 geführt hat, schränken die Verwendbarkeit des Algorithmus natürlich ein. Zum einen wird die Abschätzung durch die Annahme gleichmäßiger Verteilung der Werte in den Spalten sehr

ungenau und zum anderen könnte die Präzision durch eine genaue Berechnung der Überdeckung verbessert werden.

Um dieses Ziel einer erhöhten Genauigkeit zu erreichen, schlagen wir für baumartige Cache-Groups einen Algorithmus vor, der die Verwendung von Histogrammen nutzt.

Um den Algorithmus zu skizzieren, wollen wir in Abschnitt 3.5.1 die Formeln zusammenschreiben, die für die Berechnungen von Belang sind. In Abschnitt 3.5.2 werden wir einen einfachen Algorithmus entwickeln, um die Füllstände der Cache-Tabellen in einer baumartigen Cache-Group mit Hilfe von Histogrammen abzuschätzen. Schließlich stellen wir in Abschnitt 3.5.3 Möglichkeiten zur Optimierung des vorher vorgeschlagenen Algorithmus vor.

3.5.1 Zusammenfassung der maßgeblichen Berechnungsvorschriften

In diesem Abschnitt wollen wir die Formeln zusammenstellen, die zur Berechnung der Füllstände in baumartigen Cache-Groups benötigt werden. Wir benutzen diese Formeln in den kommenden Abschnitten 3.5.2 und 3.5.3.

Zunächst müssen wir berechnen, wie viele Sätze in die Wurzel-Tabelle geladen werden. Hierzu verwenden wir Formel 2.7.4:

$$n_{T(ck)} = \sum_{\varphi \in V_{ck}} (h_{ck}(\varphi) \cdot p_{ref}(\varphi)).$$

Dabei ist $n_{T(ck)}$ die Anzahl der in der Wurzel-Tabelle geladenen Werte. h_{ck} ist das Histogramm des Cache-Keys und p_{ref} ist die Wahrscheinlichkeit für einen Wert, referenziert zu werden.

Um die Anzahl der Sätze, die in Cache-Tabellen weiter unten im Erreichbarkeitsgraph geladen werden, berechnen zu können, verwenden wir Formel 2.8.5.3:

$$n_{T(k)} = \sum_{\varphi \in V_k} h_k(\varphi) \cdot p_{l|k}(\varphi).$$

Aus diesem Abschnitt wissen wir ebenfalls, dass wir die Ladewahrscheinlichkeit in einem Konsumenten wie folgt abschätzen können (Formel 2.8.5.4):

$$p_{l|k}(\varphi) = p_{l|b(k)}(\varphi) \cdot \chi_{b(k)\cdot\ldots\cdot k}(\varphi).$$

Dabei ist $p_{l|b(k)}(\varphi)$ die Ladewahrscheinlichkeit von φ im Basiserzeuger von k.

Wir können nun zwei Situationen unterscheiden.

- Ist der Basiserzeuger von k der Cache-Key, so gilt $p_{l|b(k)}(\varphi) = p_{ref}(\varphi)$. Einsetzen bringt dann:

$$n_{T(k)} = \sum_{\varphi \in V_k} h_k(\varphi) \cdot p_{ref}(\varphi) \cdot \chi_{b(k)\cdot\ldots\cdot k}(\varphi).$$

- Ist der Basiserzeuger von k nicht der Cache-Key, so müssen die geladenen Werte in den Basiserzeuger geschmuggelt worden sein und die Ladewahrscheinlichkeit entspricht der en Schmuggelwahrscheinlichkeit mit der endgültigen Schmuggelumgebung. Somit gilt

$$p_{l|b(k)}(\varphi) = p_{s|b(k)}(\varphi) = w_{b(k)}(h_{b(k)}(\varphi), n_{T(b(k))}).$$

Einsetzen bringt dann:

$$n_{T(k)} = \sum_{\varphi \in V_k} h_k(\varphi) \cdot w_{b(k)}(h_{b(k)}(\varphi), n_{T(b(k))}) \cdot \chi_{b(k)\cdot\ldots\cdot k}(\varphi).$$

Es ist zu beachten, dass die Cache-Tabelle des Basiserzeugers vollständig bevölkert sein muss, damit wir die Schmuggelwahrscheinlichkeit berechnen können. Denn ansonsten ist die Mächtigkeit der endgültigen Schmuggelumgebung nicht bekannt.

3.5.2 Erster Entwurf

In diesem Abschnitt wollen wir einen ersten Algorithmus vorstellen, um die Füllstände der Cache-Tabellen einer baumartigen Cache-Group mit Hilfe von Histogrammen und der Referenzierungswahrscheinlichkeit des Cache-Key-Werts abzuschätzen.

In diesem Algorithmus werden die Cache-Tabellen im Erreichbarkeitsgraph ebenfalls in topologischer Suchordnung verarbeitet. So stellen wir sicher, dass die Cache-Tabelle des Basiserzeugers b(k) jedes Konsumenten k vollständig bevölkert ist, bevor wir die Anzahl der in der Cache-Tabelle T(k) des Konsumenten k geladenen Sätze $n_{T(k)}$ abschätzen.

Wir beginnen also mit der Wurzel-Tabelle und berechnen die Anzahl der hier geladenen Sätze. Anschließend können wir die Anzahl der geladenen Sätze in den Nachfolgern berechnen usw. Abbildung 14 zeigt einen solchen Algorithmus.

```
// Berechne zunächst den Füllstand der Wurzel-Tabelle
n_T(ck) = 0;
foreach (φ ∈ V_ck) {
      n_T(ck) += h_ck(φ) * p_ref(φ);
}
// Berechne dann die Füllstände der restlichen Cache-Tabellen, betrachte
// hierfür jede Cache-Tabelle außer der Wurzel-Tabelle in TSO
foreach (T ∈ (CT - {ck}) in TSO) {
      // betrachte den Konsumenten der Cache-Tabelle
      k = k_T;
      Spalte b = finde_Basiserzeuger(k);
      n_T = 0;
      foreach (φ ∈ V_b) {
            if (χ_b·…·k(φ)) {
                  if (b == ck) {
                        n_T += h_k(φ) * p_ref(φ);
                  } else {
                        n_T += h_k(φ) * w_b(h_b(φ), n_T(b));
                  }
            }
      }
}
```

Abbildung 14: einfacher Algorithmus zur Berechnung der Füllstände mit Histogrammen

Der Algorithmus in Abbildung 14 berechnet zunächst den Füllstand der Wurzel-Tabelle. Wie immer ist ck der Cache-Key der Cache-Group. Zur Berechnung des Füllstands der Wurzel-Tabelle wird zunächst die Anzahl der dort geladenen Sätze mit 0 initialisiert. Anschließend wird iterativ jeder Wert aus der DBD des Cache-Keys betrachtet und so die durchschnittliche Anzahl der geladenen Sätze mit Hilfe der bekannten Vorschrift berechnet.

Die Füllstände der restlichen Cache-Tabellen werden in topologischer Sortierordnung berechnet.

Für jede Cache-Tabelle werden zunächst ihr Konsument und dessen Basiserzeuger bestimmt, und anschließend wird wiederum der Füllstand der Tabelle mit 0 initialisiert.

Dann werden wiederum alle Werte aus der DBD des Basiserzeugers durchlaufen. Ist die kombinierte Zugehörigkeitsfunktion für diesen Wert gleich 1, d. h. liegt der Wert in den DBDs aller Spalten auf dem WMP zwischen Basiserzeuger und Konsument, so kann der Wert im Konsument geladen sein. Die Ladewahrscheinlichkeit im Konsument entspricht dann der Ladewahrscheinlichkeit im Basiserzeuger.

Die Ladewahrscheinlichkeit im Basiserzeuger wiederum entspricht der en Schmuggelwahrscheinlichkeit, falls dieser nicht der Cache-Key ist. Ansonsten entspricht die Ladewahrscheinlichkeit im Konsumenten der Referenzierungswahrscheinlichkeit des Werts als Cache-Key-Wert. Der Erwartungswert für die Anzahl der für diesen Wert geladenen Sätze ergibt sich aus der Mächtigkeit der Werteumgebung und der Ladewahrscheinlichkeit.

Die abgeschätzte Anzahl der in die betrachtete Cache-Tabelle geladenen Sätze entspricht der Summe der Erwartungswerte für die Anzahl der geladenen Sätze für jeden betrachteten Wert. Diese Summe wird iterativ gebildet.

Zur Berechnung der Füllstände der Cache-Tabellen werden die einzelnen Tabelle in topologischer Suchordnung betrachtet. Hierdurch wird gewährleistet, dass die Cache-Tabelle des jeweilig zugehörigen Basiserzeugers vollständig bevölkert ist, so dass $n_{T(b)}$ bekannt ist und somit $w_b(h_b(\varphi), n_{T(b)})$ berechnet werden darf.

Man beachte, dass der skizzierte Algorithmus nicht besonders effizient arbeitet, da der Basiserzeuger und die kombinierte Zugehörigkeitsfunktion für jede Cache-Tabelle neue berechnet werden muss. Außerdem muss die DBD des Basiserzeugers des Konsumenten jeder Cache-Tabelle iterativ durchlaufen und jeder Wert aus der DBD betrachtet werden, um die jeweilige Schmuggelwahrscheinlichkeit und die Mächtigkeit der Werteumgebung im Konsumenten zu berechnen.

Um die Laufzeit des Algorithmus zu verbessern, können wir einige Optimierungen betrachten. Im nächsten Abschnitt wollen wir dementsprechend einen weiteren Algorithmus vorstellen, mit dem ebenfalls die Füllstände der Cache-Tabellen in einer baumartigen Cache-Group abgeschätzt werden können. Dieser Algorithmus hat eine wesentlich bessere Effizienz.

3.5.3 Steigerung der Effizienz

Wie wir gesehen haben, betrachtet der Algorithmus aus dem vorhergehenden Abschnitt für jeden Konsumenten aus der Cache-Group jeden Wert aus der DBD seines Basiserzeugers. Die komplette DBD des Basiserzeugers wird für jeden Konsumenten komplett neu durchlaufen.

Wir können nun die Effizienz steigern, indem wir für jeden Basiserzeuger in der Cache-Group die DBD nur einmal durchlaufen. Jeder dieser Werte im Basiserzeuger b trägt dann einmal dazu bei, dass der Füllstand in den Konsumenten k des Basiserzeugers b (= b(k)) steigt.

Definition: Ein **RCC-Teilbaum** ist ein Teilbaum des SBB, dessen Wurzel kein Konsument ist und dessen Knotenmenge alle Spalten umfasst, die durch über einen WMP mit der Wurzel verbunden sind. Der RCC-Teilbaum eines Basiserzeugers b ist die Vereinigung der WMPe, die bei b anfangen.

Aus der Definition ergibt sich, dass die Wurzel b eines RCC-Teilbaums der Basiserzeuger ist, der zu allen im RCC-Teilbaum liegenden Spalten gehört. Für alle anderen Spalten k in diesem RCC-Teilbaum gilt b = b(k).

Wir können nun die Füllstände in den Cache-Tabellen eines RCC-Teilbaums berechnen, indem wir über die Werte aus der DBD des Basiserzeugers iterieren und so rekursiv die Füllstände der beteiligten Cache-Tabellen gemeinsam berechnen.

```
void berechne_Füllstände(Basiserzeuger b) {
      foreach (φ ∈ V_b) {
            foreach (j ∈ {k: b→…→k}) {
                  if (χ_{b….j}(φ)) {
                        n_{T(j)} = h_j(φ) * p_{l|b}(φ);
                  }
            }
      }
}
```

Abbildung 15: Berechnung der Füllstände in einem RCC-Teilbaum

Der in Abbildung 15 gezeigte Algorithmus berechnet die Füllstände der Cache-Tabellen, deren Konsumenten im gleichen RCC-Teilbaum liegen. Die Wurzel des betrachteten RCC-Teilbaums sei der Basiserzeuger b. Wir setzen voraus, dass die Cache-Tabelle von b vollständig bevölkert ist, falls b nicht der Cache-Key ist.

Bemerkung: Wir vereinbaren, dass wir einen Basiserzeuger, dessen Cache-Tabelle vollständig bevölkert ist, in diesem Abschnitt **nutzbar** nennen. Einen Basiserzeuger, für dessen RCC-Teilbaum wir die Füllstände mit Hilfe des Algorithmus in Abbildung 15 berechnet haben, nennen wir **benutzt**.

Der Algorithmus durchläuft alle Werte φ aus der DBD von b in einer äußeren Schleife und alle Konsumenten j im betrachteten RCC-Teilbaum in einer inneren Schleife.

Falls der Wert in den DBDs aller Konsumenten zwischen b und j liegt, so kann er zu einem Laden von Sätzen in die Cache-Tabelle von j beitragen. Die Ladewahrscheinlichkeit in b ist dann gleich der Ladewahrscheinlichkeit in j.

Bemerkung: Wir zeigen in diesem Abschnitt nicht mehr explizit, wie die Ladewahrscheinlichkeit in b berechnet wird. Je nach dem, ob b der Cache-Key ist oder nicht, entspricht die Ladewahrscheinlichkeit der Referenzierungswahrscheinlichkeit oder der Schmuggelwahrscheinlichkeit. Die Ladewahrscheinlichkeit kann also wie im vorhergehenden Algorithmus (aus Abbildung 14) berechnet werden.

Die Anzahl der Sätze, die in j für φ geladen werden, ergibt sich somit als Produkt aus der Ladewahrscheinlichkeit in b und der Mächtigkeit der Werteumgebung in j.

Für die Verwendung des Algorithmus haben wir vorausgesetzt, dass die Cache-Tabelle des Basiserzeugers, der die Wurzel des RCC-Teilbaums ist, vollständig bevölkert ist oder dass b der Cache-Key ist.

Zunächst kann ohne weitere Voraussetzungen also nur der RCC-Teilbaum betrachtet werden, dessen Wurzel der Cache-Key ist. Außerdem können wir wie im vorhergehenden Abschnitt auch die Anzahl der in die Wurzel-Tabelle geladen Sätze berechnen. Damit werden auch die anderen Erzeuger der Wurzel-Tabelle als Basiserzeuger eines RCC-Teilbaums verwendbar. Durch die Berechnung der Füllstände in diesen RCC-Teilbäumen werden weitere Cache-Tabellen vollständig bevölkert und somit können weitere RCC-Teilbäume betrachtet werden usw.

Nachdem der Füllstand berechnet ist, ist eine Cache-Tabelle natürlich vollständig bevölkert. Eine Cache-Tabelle ist spätestens dann vollständig bevölkert, wenn ihr Konsument in einem RCC-Teilbaum liegt, dessen Füllstände wir bereits berechnet haben.

Wir können also eine beliebige vollständig bevölkerte Cache-Tabelle betrachten und für die Basiserzeuger, die in dieser Cache-Tabelle liegen wiederum die Füllstände des entsprechenden RCC-Teilbaums berechnen. Anschließend sind eventuell weitere Cache-Tabellen vollständig bevölkert.

Schreibweise: Wir bezeichnen die Menge der Basiserzeuger e, die dadurch nutzbar werden, dass b benutzt wurde mit $B_n(b)$.

Folgerung 25: Es gilt $e \in B_n(b)$, genau dann, wenn zwischen b und der Schmuggelquelle von e ein nicht-leerer WMP liegt.

Begründung: Der Vorgänger k von e liegt im RCC-Teilbaum von b. Da dieser RCC-Teilbaum schon betrachtet wurde, ist der Füllstand von T(k) berechnet und T(k) vollständig bevölkert. Auf der anderen Seite gilt aber T(k) = T(e), da k die Schmuggelquelle von c ist. ■

Nachdem wir den RCC-Teilbaum eines Basiserzeugers b betrachtet haben, können wir also die RCC-Teilbäume betrachten, deren Wurzel in $B_n(b)$ liegt.

Um dies zu gewährleisten, kann der Algorithmus alle Basiserzeuger in topologischer Sortierordnung betrachten.

Abbildung 16 zeigt den entwickelten Algorithmus zur Berechnung der Füllstände aller Cache-Tabellen.

```
// Betrachte jeden Basiserzeuger in topologischer Sortierordnung
foreach (b ∈ B in TOS) {
      // berechne die Füllstände der Cache-Tabellen im RCC-Teilbaum
      // unterhalb b
      berechne_Füllstände(b);
}
```

Abbildung 16: Algorithmus zur Berechnung der Füllstände aller Cache-Tabellen mit Histogrammen

4 Nicht-baumartige, RCC-zyklenfreie Cache-Groups

Im vorangegangenen Abschnitt haben wir uns ausschließlich mit baumartigen Cache-Groups beschäftigt. Diese Klasse von Cache-Groups erfüllte die Voraussetzung, dass an jeder der Cache-Tabellen maximal eine RCC endete. An der so genannten Wurzel-Tabelle, zu der der Cache-Key gehört, durfte keine RCC enden.

Wir wollen diese Voraussetzung nun etwas abschwächen und betrachten in diesem Abschnitt allgemeine, RCC-zyklenfreie Cache-Groups. D. h. wir stellen an die betrachteten Cache-Groups die Anforderung, dass an ihren Cache-Tabellen nur dann mehrere RCCs enden dürfen, wenn dadurch kein RCC-Zyklus entsteht. Wir bezeichnen zusammenhängende, geschlossene Wege, in denen auch RCC-Kanten vorkommen, als RCC-Zyklen.

Man beachte, dass es sich bei den zugrunde liegenden SBGs nicht um gerichtete, azyklische Graphen handelt, da Schmuggler-Kanten in Cache-Tabellen mit mehr als einem Konsumenten generell Zyklen bilden. Der Erreichbarkeitsgraph der Cache-Group – in dem Knoten ganze Cache-Tabellen (statt Spalten) repräsentieren – ist allerdings azyklisch.

Um diese Voraussetzung genauer zu untersuchen, betrachten wir den zur Cache-Group gehörenden Spalten-Beziehungsgraph (SBG). Dabei untersuchen wir generelle Eigenschaften von RCC-zyklenfreien Cache-Groups. Mit diesem Wissen entwickeln wir dann einen Algorithmus, um die Füllstände in den Cache-Tabellen zu berechnen. Der Algorithmus geht allerdings von der Annahme aus, dass die über verschiedene RCCs ankommenden Wertemengen stochastisch unabhängig sind. Allgemein lässt sich diese Annahme allerdings nicht halten. Dementsprechend untersuchen wir schließlich noch die Abhängigkeiten zwischen verschiedenen Wertemengen.

4.1 Der SBG nicht-baumartiger, RCC-zyklenfreier Cache-Groups

In diesem Abschnitt wollen wir grundlegende Eigenschaften von Cache-Groups untersuchen, welche die Voraussetzungen erfüllen.

Zunächst stellen wir fest, dass in den Cache-Tabellen einer nicht-baumartigen Cache-Group mehrere Konsumenten existieren können. Somit ist der Konsument einer Cache-Tabelle nicht mehr eindeutig bestimmt.

Aus der Tatsache, dass an einer Cache-Tabelle – also insbesondere auch an den Konsumenten dieser Cache-Tabelle – mehrere RCCs enden dürfen, ergibt sich im Gegensatz zu baumartigen Cache-Groups, dass für einen gegebenen Konsumenten der Basiserzeuger nicht mehr eindeutig bestimmt ist. Vielmehr gibt es für manche Konsumenten mehrere Basiserzeuger.

Der SBG einer in diesem Abschnitt 4 betrachteten Cache-Group enthält allerdings keine RCC-Zyklen. D. h. es gibt keinen zusammenhängenden Weg mit RCC-Kanten, der eine Spalte mit sich selbst verbindet, der also an der gleichen Spalte beginnt und endet. Durch diese Eigenschaft wird ein rekursives Laden im Füllzyklus ausgeschlossen und wir können die Zeitpunkte der zusammenfassenden RCC-Füllschritte in topologischer Sortierordnung der Cache-Tabellen im Erreichbarkeitsgraph der Cache-Group ausführen.

Man beachte, dass Schmuggler-Zyklen natürlich erlaubt sind. Dies ist auch gar nicht anders möglich, denn wenn wir zulassen, dass eine Cache-Tabelle mehrere Konsumenten enthalten darf, so ist jeder dieser Konsumenten sowohl eine Schmuggelquelle als auch ein Schmuggelziel. Durch diese gegenseitige Abhängigkeit ergeben sich natürlich entsprechende Schmuggler-Zyklen.

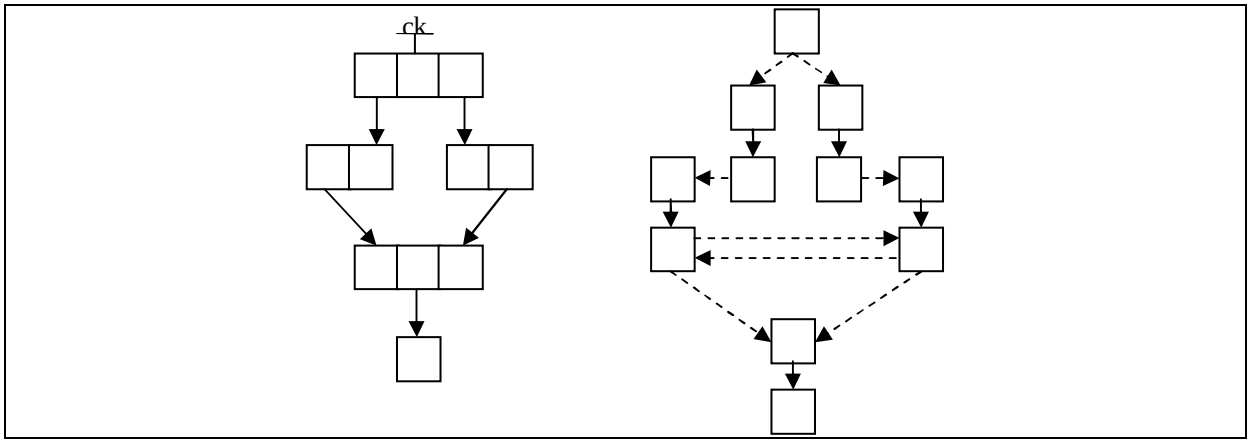

Abbildung 17: Eine nicht-baumartige, RCC-zyklenfreie Cache-Group als Erreichbarkeitsgraph und SBG

In Abbildung 17 ist eine nicht-baumartige, RCC-zyklenfreie Cache-Group dargestellt. Auf der linken Seite sieht man den Erreichbarkeitsgraph der Cache-Group. Der Erreichbarkeitsgraph ist azyklisch, wir können die Knoten also topologisch sortieren. Rechts ist die gleiche Cache-Group als SBG dargestellt. Da eine der Cache-Tabellen mehrere Konsumenten hat, existiert hier ein Schmuggler-Zyklus.

4.2 Abschätzung der Füllstände

Im folgenden Abschnitt wollen wir einen einfachen Algorithmus zur Abschätzung der Füllstände in den Cache-Tabellen einer RCC-zyklenfreien Cache-Group skizzieren. Wir verwenden hierzu zusammenfassende RCC-Füllschritte. Die Cache-Tabelle des Vorgängers einer betrachteten RCC muss vollständig bevölkert sein, damit der Füllschritt ausgeführt werden kann. Um dies zu gewährleisten, betrachten wir die Cache-Tabellen in topologischer Suchordnung. Eine solche Suchordnung ist natürlich nur möglich, wenn die betrachtete Cache-Group RCC-zyklenfrei ist. Dementsprechend kann mit dem hier vorgeschlagenen Algorithmus auch nur eine Cache-Group untersucht werden, wenn sie die Voraussetzungen, wie sie in Abschnitt 4.1 beschrieben wurden, erfüllt.

Um den Algorithmus zu entwerfen, betrachten wir zunächst einen Füllschritt in einer Cache-Tabelle. Wir können hier wieder die gleichen Fälle wie bei baumartigen Cache-Groups unterscheiden. Zum einen kann es sich bei der betrachteten Cache-Tabelle um die Wurzel-Tabelle handeln, zum anderen kann es einen nicht-leeren Weg[21] zwischen der Wurzel-Tabelle und der betrachteten Cache-Tabelle geben.

Um das Füllverhalten der Wurzel-Tabelle zu modellieren, verweisen wir auf Abschnitt 2.7. Wir verwenden in diesem Abschnitt keine Histogramme, also gehen wir von der Gleichverteilungsannahme aus. Dementsprechend können wir die Anzahl der in die Wurzel-Tabelle geladenen Werte unabhängig von der Wahrscheinlichkeit, mit der die Werte durch eine Benutzeranfrage innerhalb eines bestimmten Zeitraums referenziert wurden, abschätzen. Wir verwenden hierfür die folgende Formel (Aufgrund von Folgerung 2, Seite 50):

$$n_{T(ck)} = |S_{T(ck)}| / c_{ck}.$$

Betrachten wir nun das Füllverhalten von Cache-Tabellen unterhalb der Wurzel.

Die betrachtete Cache-Tabelle T habe ohne Beschränkung der Allgemeinheit $r \geq 1$ eingehende RCCs an eventuell verschiedenen Konsumenten. Wir gehen davon aus, dass die Cache-Tabelle $1 \leq j \leq r$ Konsumenten hat.

Für jede der eingehenden RCCs wird ein zusammenfassender RCC-Füllschritt aufgeführt. Die Cache-Tabelle am Anfang der RCC muss hierfür vollständig bevölkert sein, was durch die Betrachtung der Cache-Tabellen in topologischer Suchordnung garantiert wird. Durch diese Voraussetzung steht zum Zeitpunkt eines Füllschritts für den Erzeuger am Anfang der betrachteten RCC fest, wie viele Werte in ihn geladen wurden.

Betrachten wir nun einen solchen zusammenfassenden RCC-Füllschritt. Die betrachtete RCC sei $e_i \rightarrow k_i$ (mit $1 \leq i \leq r$ als Nummer der betrachteten RCC).

Mit Hilfe von Folgerung 9 kann die Anzahl der über diese RCC induzierten Werte berechnet werden. Es gilt:

$$v_{i|e_i \rightarrow k_i} = v_{l|e_i} \cdot \ddot{u}_{k_i, e_i}.$$

Zur Berechnung der Anzahl der über die betrachtete RCC induzierten Werte müssen wir also die Anzahl $v_{l|e_i}$ der in den Erzeuger am Anfang der RCC geladenen Werte bestimmen. Wie wir aus Abschnitt 2.8 wissen, können wir diese Anzahl $n_{T(e_i)}$ aus der Anzahl der in die Cache-Tabelle $T(e_i)$ geladenen Sätze ableiten:

$$v_{l|e_i} = f_{e_i}(n_{T(e_i)}).$$

In diese Überlegung geht implizit die Annahme ein, dass alle Werte in den Erzeuger e_i geschmuggelt werden. Wie wir aus Abschnitt 2.8 wissen, tragen aber nur die Sätze aus der endgültigen Schmuggelumgebung zum Schmuggeln von Werten bei, und nicht alle geladen Sätze. Somit ist im Allgemeinen nur ein Teil der geladenen Sätze geschmuggelt, der Rest ist

[21] Ein nicht-leerer Weg ist ein Weg mit Länge größer 0.

induziert. Von diesem Sachverhalt wird in den Berechnungen in diesem Abschnitt abstrahiert, um den Algorithmus so einfach wie möglich zu halten.

Jetzt können wir die Mächtigkeit der Werteumgebung der über die RCC $e_i \rightarrow k_i$ (im Konsument k_i) induzierten Werte abschätzen:

$$s_{i|e_i \rightarrow k_i} = v_{i|e_i \rightarrow k_i} \cdot |S_T| / c_{k_i} .$$

Durch Einsetzen erhalten wir:

$$s_{i|e_i \rightarrow k_i} = f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i, e_i} \cdot |S_T| / c_{k_i} .$$

Die Abschätzung ist auf die Art möglich, da wir von der Gleichverteilungsannahme ausgehen und dementsprechend für jeden Wert $|S_T| / c_{k_i}$ Sätze in der Cache-Tabelle geladen sind.

Wir können nun die Anzahl der insgesamt geladenen Sätze (Menge $S_{l|T}$) abschätzen, indem wir die Mächtigkeit der Vereinigung der Werteumgebung der über die verschiedenen RCCs induzierten Werte betrachten. Für die Vereinigung gilt formal:

$$S_{l|T} = \bigcup_{i=1}^{r} S_{i|e_i \rightarrow k_i} .$$

Dabei gehen wir davon aus, dass die betrachteten RCCs von 1 bis r durchnummeriert sind. Die Vereinigung können wir nun umformen, um sie iterativ zu berechnen. Es gilt:

$$\bigcup_{i=1}^{r} S_{i|e_i \rightarrow k_i} = \underbrace{(...(S_{i|e_1 \rightarrow k_1} \cup ...) \cup S_{i|e_i \rightarrow k_i} ...)}_{:= S_{l|T|\tau_T + i + 1}} \cup S_{i|e_r \rightarrow k_r}$$

Dabei ist $S_{l|T|\tau_T+i}$ die Menge der nach dem i-ten, durch die Cache-Tabelle T ausgeführten, Füllschritt geladenen Sätze. Man beachte, dass wir die einzelnen Cache-Tabellen in topologischer Sortierordnung des Erreichbarkeitsgraphen betrachten. Somit führt eine Cache-Tabelle alle ihre Füllschritte hintereinander aus. τ_T+i ist dementsprechend der Zeitpunkt des Füllschritts, bei dem die i-te RCC in Cache-Tabelle T betrachtet wird.[22] Wir bezeichnen den Zeitpunkt unmittelbar vor dem ersten Füllschritt der Cache-Tabelle T mit τ_T.

Wir können die Menge der geladenen Sätze nun iterativ bilden. Es gilt:

$$S_{l|T|\tau_T} = \varnothing;\ S_{l|T|\tau_T+i+1} = S_{l|T|\tau_T+i} \cup S_{i|e_i \rightarrow k_i} .$$

Da wir keine Information über Zusammenhänge zwischen den Mengen der geladenen Werte in den Erzeugern am Anfang verschiedener RCCs haben, nehmen wir prinzipiell die Unabhängigkeit der Wertemengen an[23]. Wir gehen also davon aus, dass keine stochastischen Abhängigkeiten zwischen den Wertemengen existieren.

---

[22] Wir nummerieren die Zeitpunkte also einfach durch.

[23] Wie realistisch diese Annahme ist, hängt vollkommen von der konkret betrachteten Cache-Group ab. Die Annahme muss aber getroffen werden, damit das Modell nicht übermäßig komplex wird.

Wir können nun versuchen, die Mächtigkeit der Menge $S_{l|T|\tau_T+i+1}$ anzunähern, indem wir Formel 2.5.1 anwenden. Wir verwenden die gemeinsame Obermenge S_T der im Backend für T geladenen Sätze.[24] Somit können wir rechnen:

$$s_{l|T|\tau_T} = 0;\ s_{l|T|\tau_T+i+1} = s_{l|T|\tau_T+i} + s_{i|e_i \to k_i} - s_{l|T|\tau_T+i} \cdot s_{i|e_i \to k_i} \ / |S_T| =$$

$$s_{l|T|\tau_T+i} + f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i,e_i} \cdot |S_T| / c_{k_i} - s_{l|T|\tau_T+i} \cdot f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i,e_i} \cdot |S_T| / c_{k_i} \ / |S_T| =$$

$$s_{l|T|\tau_T+i} + |S_T| \cdot f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i,e_i} / c_{k_i} - s_{l|T|\tau_T+i} \cdot f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i,e_i} / c_{k_i} =$$

$$s_{l|T|\tau_T+i} + (|S_T| - s_{l|T|\tau_T+i}) \cdot f_{e_i}(n_{T(e_i)}) \cdot \ddot{u}_{k_i,e_i} / c_{k_i}.$$

Wir können nun den Algorithmus skizzieren, um den Füllstand einer Cache-Tabelle abzuschätzen.

```
void berechne_Füllstand_zusammenfassende_RCC_Füllschritte(Cache-Tabelle T)
{
      // Wir nehmen an, dass die Cache-Tabelle vor dem ersten Füllschritt
      // leer war
      n_T = 0;
      // Für jeden Füllschritt mit e als Erzeuger und k als Konsument…
      foreach (e→k mit k ∈ K_T) {
            // Berechne die Anzahl der nun geladenen Sätze
            n_T = n_T+(|S_T|-n_T)·f_e(n_T(e))·ü_k,e/c_k;
      }
}
```

Abbildung 18: Algorithmus zur Berechnung des Füllstands einer Cache-Tabelle mit zusammenfassenden RCC-Füllschritten

Die in Abbildung 18 gezeigte Funktion berechnet den Füllstand einer Cache-Tabelle mit Hilfe von zusammenfassenden RCC-Füllschritten iterativ. Zunächst wird der Füllstand mit 0 initialisiert, da wir annehmen, dass die Cache-Tabelle vor der Betrachtung, also vor dem ersten Füllschritt, leer war. Anschließend wird in einer Schleife jede bei der Cache-Tabelle eingehende RCC betrachtet und der Füllstand gemäß der hergeleiteten Vorschrift erhöht.

Alternativ zu dem betrachteten Verfahren könnte man auch eine Worst-Case-Analyse machen, indem man folgende Annahmen trifft:

- Jeder der geladenen Sätze trägt zu einem Schmuggeln von Werten in den Erzeuger am Anfang der RCC bei,
- jeder ankommende Wert liegt in der DBD des betrachteten Konsumenten, und
- die Schnittmengen der Werteumgebungen der über die verschiedenen RCCs geladenen Werte sind disjunkt.

Unter diesen Bedingungen würde sich folgende Berechnungsvorschrift ergeben:

[24] Man beachte, dass die Abschätzung besser würde, wenn wir eine „engere“ gemeinsame Obermenge kennen würden. Ein solches Wissen steht uns aber hier nicht zur Verfügung.

$$s_{l|T|\tau_T} = 0 \qquad s_{l|T|\tau_T+i+1} = \min(s_{l|T|\tau_T+i} + s_{i|e_i \to k_i}, |S_T|) = \min(s_{l|T|\tau_T+i} + \min(s_{l|T(e_i)}, c_{e_i} \cdot \ddot{u}_{e_i,k_i}), |S_T|)$$

Die beiden Minimum-Funktionen lassen sich folgendermaßen erklären: Die Anzahl der insgesamt geladenen Sätze ist nach oben um die Anzahl der im Backend vorhandenen Sätze begrenzt, und die Anzahl der ankommenden Werte wird durch die Kardinalität der Schnittmenge zwischen dem betrachteten Erzeuger und Konsumenten beschränkt.

Indem wir alle Cache-Tabellen in topologischer Suchordnung betrachten, können wir alle Füllstände berechnen.

```
// Betrachte jede Cache-Tabelle in topologischer Suchordnung
foreach (T ∈ CT in TOS) {
      // Wenn es in T keine Konsumenten gibt, handelt es sich um die
      // Wurzel-Tabelle
      if (K_T = ∅) {
            // Berechne die Anzahl der Sätze, wenn ein Cache-Key-Wert in
            // den Cache-Key ck geladen wird
            n_T = N_T/c_ck;
      // Ansonsten haben wir eine Cache-Tabelle unterhalb der Wurzel-
Tabelle
      } else {
            // Berechne den Füllstand wie gehabt.
            berechne_Füllstand_zusammenfassende_RCC_Füllschritte(T);
      }
}
```

Abbildung 19: Algorithmus zur Berechnung aller Füllstände

Abbildung 19 zeigt einen Algorithmus zur Berechnung der Füllstände aller Cache-Tabellen in einer Cache-Group. Die Cache-Tabellen werden in topologischer Suchordnung durchlaufen und die Füllstände der Cache-Tabellen unterhalb der Wurzel gemäß der oben beschriebenen Berechnungsvorschrift berechnet. Für die Wurzel-Tabelle wird mit Hilfe der bekannten Formel wie bei baumartigen Cache-Groups der Füllstand nach dem Laden eines Cache-Key-Werts berechnet.

4.3 Schwachpunkte des Algorithmus

Der Algorithmus hat mehrere Schwachpunkte, die wir in diesem Abschnitt betrachten wollen. Die Ungenauigkeiten des in Abschnitt 4.2 vorgestellten Algorithmus haben folgende Gründe:

- Wir vernachlässigen die Abhängigkeiten zwischen ankommenden Wertemengen. Wir verwenden also die Spaltenunabhängigkeitsannahme und sehen das Laden von Werten über verschiedene RCCs als unabhängige Ereignisse. Das liegt daran, dass wir die Cache-Tabellen in topologischer Sortierordnung betrachten und diese zusammenfassende RCC-Füllschritte ausführen. Wir können also nicht identifizierten, wenn zwei zu ladende Werte identisch sind, weil sie z. B. aus dem gleichen Basiserzeuger stammen, aber über unterschiedliche WMPe angekommen sind.
- Wir verwenden die Gleichverteilungsannahme, vernachlässigen also unterschiedliche Verteilung der Werte in den Spalten. Je nach dem, wie weit die tatsächliche Verteilung

von der Gleichverteilung abweicht, kann es hierdurch bei der Berechnung der Anzahl der geladenen Sätze zu Ungenauigkeiten kommen.

- Durch das Zurückrechnen von der Anzahl der geladenen Sätze auf die Anzahl der geladenen Werte in einem Erzeuger wird eine Ungenauigkeit eingeführt.

Die Ungenauigkeiten betreffen die Berechnung der Anzahl der in jeweils eine betrachtete Cache-Tabelle geladenen Sätze. Diese Ungenauigkeiten pflanzen sich jedoch im Laufe des Algorithmus über die verschiedenen Cache-Tabellen fort. Prinzipiell ist somit auch ein Aufschaukeln der Ungenauigkeiten möglich.

Eine quantitative Analyse der Ungenauigkeit, die durch die Vernachlässigung der Abhängigkeiten oder die Anwendung der Gleichverteilungsannahme entsteht ist allgemein nicht möglich, da dies sehr von der konkret betrachteten Cache-Group abhängt. Die Betrachtung der verschiedenen Details, die hier auf die Ungenauigkeit Einfluss haben, würde den Rahmen dieses Abschnitts sprengen. Wir wollen aber zumindest festhalten, dass die Vernachlässigung der Abhängigkeiten allgemein einen großen Einfluss auf das Ergebnis der Berechnung hat. Aus diesem Grund schlagen wir die Verwendung des Algorithmus für allgemeine Cache-Groups und -Föderationen, wie er in Kapitel 5 vorgestellt wird, auch zur Berechnung der Füllstände in nicht-baumartigen, RCC-zyklenfreien Cache-Groups vor.

Im Folgenden wollen wir die Ungenauigkeit betrachten, die durch das Zurückrechnen von der abgeschätzten Anzahl der geladenen Sätze auf die Anzahl der in einen Erzeuger geladen Sätze entsteht. Wir nehmen nämlich implizit für den Algorithmus an, dass alle Werte in die Erzeuger einer Cache-Tabelle geschmuggelt wurden.

Beispiel: Betrachten wir einen Konsumenten k, der gleichzeitig Erzeuger ist. Der Einfachheit halber setzen wir voraus, dass diese Spalte der einzige Konsument ihrer Cache-Tabelle sei und k nur eine ankommende RCC e→k habe. Aus dieser Voraussetzung ergibt sich, dass die RCC isoliert ist und dementsprechend alle nach dem Füllzyklus in k geladenen Werte induziert und umgebungsvollständig sind.

Der Algorithmus führt für die betrachtete Cache-Tabelle, die den Anforderungen aus dem hier vorgestellten Beispiel genügt, im Prinzip drei Rechenschritte aus:

- Abschätzung der Anzahl der in k induzierten Werte
- Abschätzung der Anzahl der in T(k) geladenen Sätze
- Abschätzung der Anzahl der in k geladenen Werte aufgrund der Anzahl der geladenen Sätze

Betrachten wir diese drei Rechenschritte im Detail. Wir können die ersten beiden Schritte zusammenfassen. Somit wird zunächst für die Cache-Tabelle T(k) folgende Berechnung durchgeführt, um die Anzahl der in T(k) geladen Werte abzuschätzen:

$$n_{T(k)} = v_{l|e} \cdot \ddot{u}_{k,e} \cdot |S_{T(k)}| / c_k.$$

Anschließend betrachtet der Algorithmus diese Spalte als Erzeuger und berechnet die Anzahl geladenen Werte wird nun wiederum mit der folgenden Formel:

$$v_{l|k} = f_k(n_{T(k)}).$$

Hier geht die Annahme ein, dass alle Werte in die Erzeuger geschmuggelt werden. Wir nehmen also implizit an, dass $v_{l|s} = v_{s|s}$ gilt.

Wir wissen aber, dass die in k geladenen Werte induziert wurden und sich ihre Anzahl genauer direkt abschätzen lässt:

$$v_{l|k} = v_{l|e} \cdot \ddot{u}_{k,e}.$$

Durch die Abschätzung der Anzahl der in die verschiedenen Erzeuger geladenen Werte ergibt sich also insgesamt eine Ungenauigkeit, die wie folgt berechnet werden kann:

$$u_k = f_k(v_{l|e} \cdot \ddot{u}_{k,e} \cdot |S_{T(k)}| / c_k) - v_{l|e} \cdot \ddot{u}_{k,e}$$

Dabei ist u_k die Ungenauigkeit, die dadurch entsteht, dass implizit angenommen wird, dass alle Werte in k geschmuggelt wurden, also durch das Zurückrechnen von der Anzahl geladener Sätze auf die Anzahl geladener Werte in k.

Zusammenfassend können wir sagen, dass sich ein Algorithmus wie er in diesem Abschnitt vorgeschlagen wurde, im Allgemeinen nicht anbietet. Zum einen ist die Menge der Cache-Groups, die die Voraussetzungen erfüllt noch recht klein. Zum anderen hat der Algorithmus viele Ungenauigkeiten.

Für allgemeine Cache-Groups und -Föderationen schlagen wir eine Modellierung wie im nächsten Abschnitt vor. Auf diese Art kann auch das Füllverhalten von nicht-baumartigen, RCC-zyklenfreien Cache-Groups betrachtet werden.

5 Allgemeine Cache-Groups und -Föderationen

Wie wir festgestellt haben, ist das einfache Modell zur Analyse des Füllverhaltens von Cache-Groups nur in einzelnen, sehr speziellen Fällen von Cache-Groups ausreichend. Zum einen können wir damit keine Aussage über das Füllverhalten von Cache-Groups mit Zyklen treffen. Zum anderen können wir das Laden in Tabellen, an denen mehrere RCCs enden, nur unter der Annahme analysieren, dass RCC-Füllschritte stochastisch unabhängige Ereignisse darstellen. Diese Annahme kann leider im Allgemeinen nicht gehalten werden.

Wir benötigen also ein verfeinertes Modell, mit dem allgemeine Cache-Groups und sogar Cache-Föderationen bewertet werden können. In diesem Abschnitt wollen wir ein Ansatz für ein Modell beschreiben, um zyklenbehaftete Cache-Groups und Cache-Föderationen bewerten zu können. Weil es die Berechnungen stark vereinfacht und dennoch einen guten Hinweis dafür geben kann, ob eine Cache-Group bzw. -Föderation ungünstige Eigenschaften hat, schlagen wir ein Modell für die Worst-Case-Analyse vor. Dieses Modell soll als Ergebnis liefern, wie viele Sätze maximal in der betrachteten Cache-Group sind, wenn wir einen einzelnen Füllzyklus betrachten.

Um das Füllverhalten der Cache-Group zu bewerten, schlagen wir vor, dass der Algorithmus für jeden Cache-Key einmal ausgeführt wird und die sich ergebenden Füllstände in den verschiedenen Cache-Tabellen gemittelt werden.

Um die Komplexität des Modells so gering wie möglich zu halten, verwendeten wir die in Abschnitt 2.1 vorgestellten Modellannahmen. Wir gehen also auf Grund der Gleichverteilungsannahme davon aus, dass jeder Wert aus der DBD einer Spalte gleich oft in der Spalte vorkommt. Außerdem nehmen wir an, dass die Verteilung der Werte in den Spalten voneinander unabhängig ist, d. h. wir verwenden die Spaltenunabhängigkeitsannahme. Schließlich abstrahierten wir von Null-Werten in den Spalten.

5.1 Token und Stellen

Der „Fluss“ der Werte in der Cache-Group bzw. -Föderation wird in diesem Modell über **Token** modelliert. Durch die im Token gespeicherten Daten soll unter anderem festgestellt werden, woher die ankommenden Werte stammen und welchen Weg sie genommen haben. In dem hier vorgeschlagenen Modell repräsentieren die Token Wertemengen. Wir werden uns in Abschnitt 5.2 mit der Semantik der Token genauer auseinander setzen.

Um die Position der Token zwischen den durch RCCs verbunden Spalten festzuhalten, verwendeten wir **Stellen**. Eine Stelle kann eine endliche Anzahl von Token aufnehmen und für den Füllschritt, bei der die Spalte am Ende der RCC aktiv wird, vorhalten.

Wir haben nun die notwendigen Mittel kennen gelernt, um Netze zur Modellierung von Cache-Groups aufzubauen. Ein solches Netz nennen wir Spalten-Beziehungsnetz (SBN).

Definition: Ein **Spalten-Beziehungsnetz** (SBN) entsteht aus einem Spalten-Beziehungsgraph, indem die RCC-Kanten durch eine Folge Kante, Stelle, Kante ersetzt werden. Zusätzlich wird zu jedem Cache-Key eine Kontrollspalte eingefügt, die über eine Folge Kante, Stelle, Kante mit dem Cache-Key verbunden ist.

Schmuggler-Kanten aus dem SBG bleiben auch im SBN bestehen, erhalten aber keine zugehörige Stelle.

Durch Einfügen der zusätzlichen Kontrollspalten und Stellen wird die Füllpunkteigenschaft der Cache-Keys modelliert. Ein so genanntes **initiales Token** in dieser Stelle repräsentiert einen neuen Cache-Key-Wert, der „von außen“ einen einzelnen Füllzyklus anstößt. Die Einstiegspunkteigenschaft bei Nicht-Unique-Cache-Keys wird durch eine Selbst-RCC (mit entsprechender Stelle) modelliert.

Inspiration für dieses Modell waren Petrinetze, bei denen ebenso Token in Stellen gespeichert und in Transitionen verarbeitet werden. Wir können die Spalten im SBN also auch als Transitionen sehen. Die Auswertung des Modells entspricht dann der „Simulation“ des Netzes.

Allerdings gibt es prinzipielle Unterschiede zwischen Petrinetzen und dem hier vorstellten SBN als Token-Stellen-System: Unter anderem sind die Transitionen in einem Petrinetz zustandslos, verändern ihr Verhalten also nicht auf Grund der vergangenen Verarbeitungen. In unserem SBN hängt das Verhalten einer Spalte aber unter anderem von der in ihr gespeicherten Statistik ab. Unter der **Statistik** fassen wir die Informationen über die Anzahl der umgebungsvollständigen und umgebungsfragmentarischen Werte sowie den Füllstand der zugehörigen Cache-Tabelle zusammen.

Ein Token wird immer in einem Erzeuger generiert, wenn in ihn neue Werte geschmuggelt werden. Dieses Token wird dann bei der Ausführung von Füllschritten in den Konsumenten verarbeitet und eventuell weiter gereicht, wenn der Konsument auch ein Erzeuger ist. Ein Token wandert also immer entlang eines WMP. Hat ein Konsument mehrere ausgehende RCCs, so wird das Token mit allen in ihm gespeicherten Daten dupliziert (geklont) und die Duplikate werden über die ausgehenden RCCs weiter gegeben. In Konsumenten ohne ausgehende RCCs werden verarbeitete Token nach einem Füllschritt **verworfen**, d. h. gelöscht. Der Zeitraum von der Generierung eines Tokens bis zum Zeitpunkt des Füllschritts, nach dem das Token verworfen wird, nennen wir **Lebenszeit** des Tokens.

Um den Fluss der Token zu überwachen, verwendeten wir einen **Token-Manager**.[25] Diese Modellkomponente überprüft nach jedem Füllschritt, wo Token in Stellen gespeichert sind, entnimmt eines und „benachrichtigt“ die Spalte am Ende der RCCs, so dass diese aktiv wird

[25] Eigentlich ist dies bereits bei Aspekt der Implementierung. Dennoch führen wir diese Komponente auf Grund der besseren Verständlichkeit des Modells hier ein.

und ihre Cache-Tabelle einen Füllschritt ausführt. Wir sagen, das Token **kommt** dann beim ausgewählten Konsumenten **an**. Analog sprechen wir von ankommenden Werten, wenn wir die Werte bezeichnen wollen, die dass ankommende Token repräsentiert. Die Spalte am Ende der RCC, die durch die Stelle repräsentiert wird, von der das Token entnommen wird, wird **aktiv** und verarbeitet das Token (was dem Induzieren von Werten über den Wirkzusammenhang I entspricht – auch in diesem Kontext wird die Spalte aktiv genannt, siehe Abschnitt 1.7).

Prinzipiell werden solange Füllschritte ausgeführt, bis keine Token mehr zirkulieren. In dieser Situation kann keine Spalte mehr aktiv werden und somit werden keine weiteren Füllschritte ausgeführt. Der Füllzyklus ist also beendet.

Da zu einem gegebenen Zeitpunkt mehrere Token in verschiedenen Stellen gespeichert sein können, kann der Token-Manager eine Auswahl treffen, welches Token im nächsten Füllschritt verarbeitet werden soll. Hierfür gibt es natürlich mehrere Möglichkeiten:

- Das Token, dass am längsten nicht verarbeitet wurde, wird gewählt oder
- das Token auf dem kürzesten WMP (das durchschnittlich also nur noch kurz vorhanden ist) wird verarbeitet oder
- es wird zufällig ein Token zur Verarbeitung ausgewählt.

Die Liste ließe sich beliebig erweitern, des Weiteren sind auch Kombinationen möglich (z. B. kann es mehrere Token mit gleichem „Alter“ geben, dann kann unter diesen eins zufällig ausgewählt werden. Es ist noch eine offene Frage, welche Serialisierung (darum handelt es sich im Endeffekt) das genaueste Ergebnis liefert. Wir gehen im Folgenden von einer zufälligen Wahl aus. Der vorgeschlagene Algorithmus arbeitet also nicht-deterministisch. Wir können das Modell dementsprechend auch in die Klasse der so genannten „Monte-Carlo-Algorithmen“ einordnen.

Abbildung 20 (nächste Seite) zeigt den Fluss eines Tokens durch einen Ausschnitt eines SBN. Mehrere Spalten sind über RCCs miteinander verbunden (modelliert durch eine Folge Kante, Stelle, Kante). Die Spalten sind dabei allgemein auch Schmuggelquellen, was durch abgehende gestrichelte Pfeile dargestellt ist. Lediglich c_1 ist ein Schmuggelziel. Zum Zeitpunkt τ_1 führt die Cache-Tabelle dieser Spalte einen Füllschritt aus und dadurch werden zusätzliche Werte in c_1 eingeführt. Aus diesem Grund wird ein neues Token generiert und in die abgehende Stelle zwischen c_1 und c_2 eingefügt. Das Token wandert nun über die Wertemengen-Pfade und wird dabei in Spalten mit mehreren ausgehenden RCCs dupliziert. Zum letzten dargestellten Zeitpunkt τ_9 existieren vier Token, die alle durch Duplikation des ursprünglichen Tokens entstanden sind.

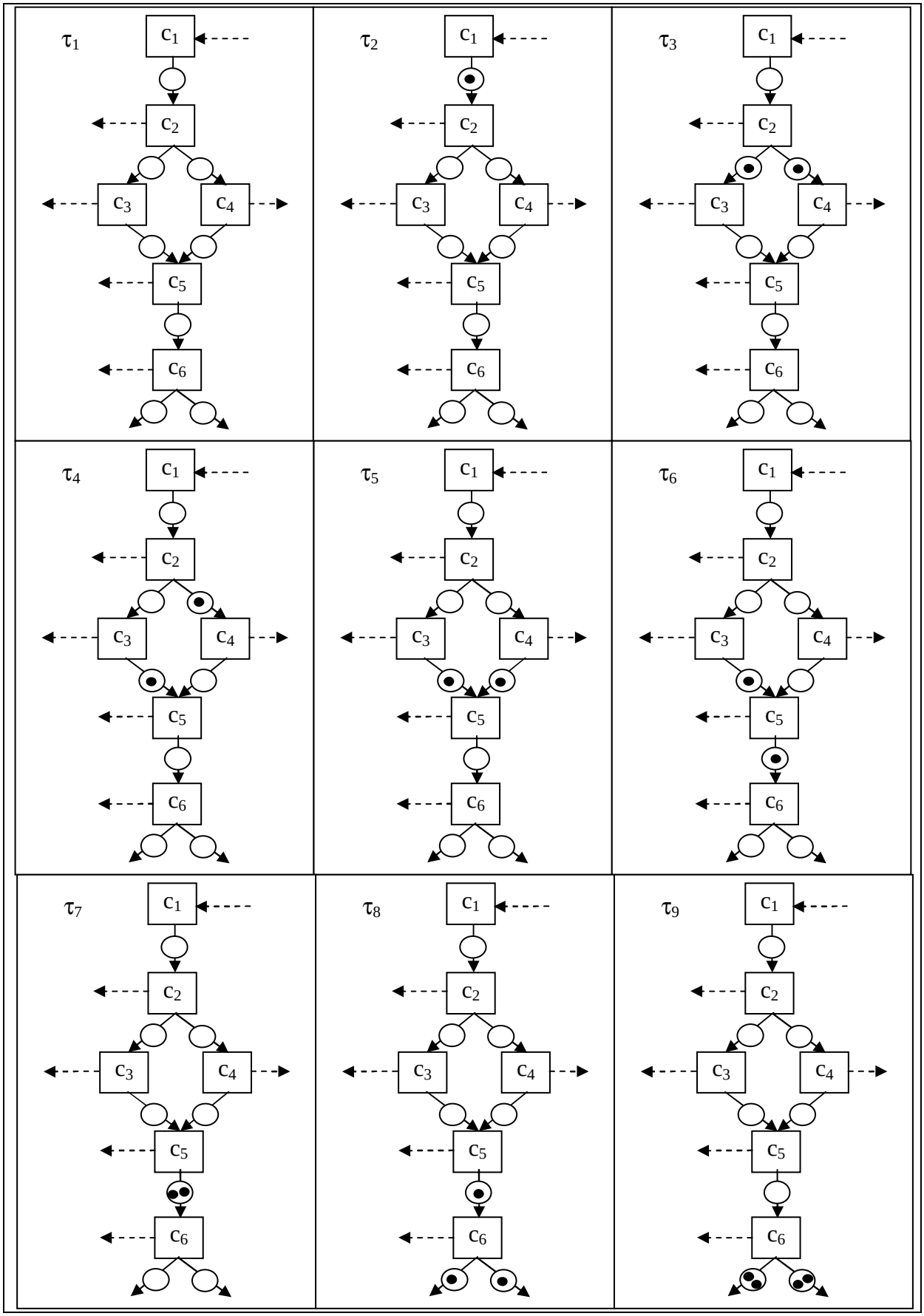

Abbildung 20: Fluss von Token mit mehrfacher Duplikation innerhalb der Verarbeitung.

5.2 Semantik der Token

Bisher haben wir zwar Entstehung, Verarbeitung, Fluss und Verwerfen von Token diskutiert, aber noch nicht geklärt, welche Semantik die Token eigentlich haben. Am einfachsten wäre es sicherlich, wenn ein Token einen einzelnen Wert repräsentieren würde. Diese Semantik würde einer Betrachtung von Einzelwert-Füllschritten entsprechen. Ein Token mit dieser Semantik würde also dann generiert, wenn der Wert zusätzlich in einen Erzeuger gelangt. Das Token würde dann entlang der RCCs immer dann weiter gereicht, wenn der korrespondierende Wert zusätzlich in einen Konsumenten geladen wird.

Bei großen Tabellen mit vielen Werten in den Spalten kommt es bei dieser Semantik aber schnell zu einem sehr hohen Aufkommen an zirkulierenden Token, was wiederum negative Auswirkungen auf die Laufzeit des Algorithmus zur Berechnung der Füllstände nach sich zieht. Um dieser Problematik zu entgegnen, schlagen wir die Betrachtung von direkten RCC-Füllschritten vor. Mit dieser Semantik repräsentieren Token also nicht mehr einzelne Werte sondern Wertemengen.

Die Token zeichnen auf ihrem Weg über die Wertemengen-Pfade die Anzahl der Werte, die bei den verschiedenen Spalten entlang des WMP ankommen, auf. Wir betrachten dies genauer in Abschnitt 5.2.1. Man beachte, dass es auch möglich wäre, dass die Token lediglich Informationen über die aktuelle Anzahl der repräsentierten Werte und die Herkunft der Werte (ursprüngliche Spalte) speichern. Dann wäre es allerdings nicht mehr möglich, Abhängigkeiten zwischen Wertemengen (Überschneidungen) auf Grund der gespeicherten Daten festzustellen.

5.2.1 Besuche und Historien

Besuche unterteilen Füllschritte in kleinere Einheiten. Während eines Besuchs gelangen neue Werte in eine Spalte. Ein Besuch wird durch die Verarbeitung eines Tokens repräsentiert. Um uns näher mit dem Thema beschäftigen zu können, benötigen wir zunächst ein paar Begriffe.

- Ein **manifestierter Besuch**[26] ist ein Paar $(c, V'_{l|c|\tau})$, wobei c eine Spalte und $V'_{l|c|\tau}$ die Menge der in Spalte c zum Zeitpunkt τ zusätzlich geladenen Werte ist. Die Spalte im Besuch heißt **besuchte Spalte**, der Zeitpunkt τ heißt **Besuchszeitpunkt** und die Wertemenge $V'_{l|c|\tau}$ eines manifestierten Besuchs heißt **besuchende Wertemenge**.
- Ein **abstrahierter Besuch** ist ein Paar $(c, v'_{l|c|\tau})$, wobei c wie beim manifestierten Besuch eine Spalte ist und $v'_{l|c|\tau}$ die Mächtigkeit der Wertemenge $V'_{l|c|\tau}$ abschätzt.

Wir können uns einen Besuch bildhaft so vorstellen, dass die Werte die Spalte besuchen und dort verarbeitet werden. Entweder es handelt sich dabei um ankommende Werte (in diesem

[26] Allgemein bezeichnet man als Manifestation Dinge, die sichtbar bzw. erkennbar werden. Unsere manifestierten Besuche offenbaren die während des Füllschritts in die besuchte Spalte zusätzlich geladenen Werte. Der zugehörige abstrahierte Besuch lässt dagegen lediglich die Mächtigkeit der Menge der zusätzlich geladenen Werte erkennen.

Fall ist die Spalte aktiv) oder die Werte werden in einen Erzeuger geschmuggelt und fließen von dort aus weiter über WMPe.

In den Modellen zur Berechnung der Füllstände werden lediglich abstrahierte Besuche eingesetzt. Um diese Modelle beschreiben zu können und deren Arbeitsweise zu erklären, benutzen wir manifestierte Besuche.

Beispiel: Die Cache-Tabelle T habe zwei Spalten: k sei ein Konsument und e ein Erzeuger. Zum Zeitpunkt τ führe T einen Füllschritt aus. Hierdurch entstehen zwei Besuche: Zum einen werden Werte in den aktiven Konsumenten k geladen. Dieses Ereignis halten wir im Besuch $(k, V'_{i|k|\tau})$ fest. Zum anderen werden aber gleichzeitig Werte in den Erzeuger e als Schmuggelziel geschmuggelt, wobei k die Schmuggelquelle ist. Wie wir gesehen haben ist k während des Füllschritts die aktive Spalte. Somit entsteht der zweite Besuch $(e, V'_{s|e|\tau})$.

Wird bei einem Besuch ein neues Token generiert, so heißt dieser Besuch **Erzeugung**. In diesem Fall nennen wir den Besuchszeitpunkt **Erzeugungszeitpunkt** und die besuchte Spalte **Erzeugungsspalte** (des Tokens). Grundsätzlich geschieht dies immer, wenn neue Werte in einen Erzeuger geschmuggelt werden. Aber nicht nur bei Besuchen werden Token erzeugt. Wir können zwei Arten von Erzeugungen unterscheiden:

- Es wird einmal vor dem Füllzyklus ein initiales Token erzeugt und in die Stelle zwischen dem betrachteten Cache-Key und seiner Kontrollspalte eingefügt. Durch die Verarbeitung dieses Tokens wird der Füllzyklus angestoßen. Allerdings gibt es bei dieser **initialen Erzeugung** keine (in der Cache-Group existierende) besuchte Spalte, da der Wert „von außen“ kommt. Wir modellieren dies, indem wir für die besuchte Spalte die Kontrollspalte des Cache-Keys einsetzen. Der Besuchszeitpunkt ist der Zeitpunkt α vor dem ersten Füllschritt. Die besuchende Wertemenge besteht lediglich aus dem neuen Cache-Key-Wert.
- Immer wenn zusätzliche Werte in Erzeuger geschmuggelt werden, wird ein Token generiert. Die besuchte Spalte ist dann der Erzeuger, wobei diese nicht aktiv ist. Die besuchende Wertemenge ist die Menge der zusätzlich in den Erzeuger geschmuggelten Werte und der Besuchszeitpunkt ist der Zeitpunkt des Füllschritts.

Ein Token speichert immer eine Kette $\mathfrak{B}$ von (abstrahierten) Besuchen und den Zeitpunkt τ_1 seiner Erzeugung. Wir nennen diese Daten **Historie** des Tokens. Formal ist eine Historie ein Paar $(\tau_1, \mathfrak{B})$ aus einem Erzeugungszeitpunkt τ_1 und einer **Besuchskette**

$$\mathfrak{B} = (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow \ldots \rightarrow (c_n,\ v'_{l|c_n|\tau_n})$$

Natürlich entspricht – außer bei der Erzeugung – die Menge der während eines Besuchs in die besuchte Spalte zusätzlich geladenen Werte den dort zusätzlich indizierten Werten, denn auf Grund der Verarbeitung des Tokens war die Spalte aktiv:

$$V'_{i|c_i|\tau_i} = V'_{l|c_i|\tau_i} \text{ für } 2 \le i \le n.$$

Aus Gründen der Einheitlichkeit schreiben wir trotzdem für die besuchende Wertemenge $V'_{l|c_i|\tau_i}$.

Wir können nun die Historie $(\tau, \mathfrak{B})$ folgendermaßen iterativ bilden:

- Der Anfang der Kette $\mathfrak{B}$ ist eine Erzeugung $(e, v'_{l|e|\tau_e})$ und
- bei jeder Verarbeitung des Tokens bei einem Füllschritt zum Zeitpunkt τ_i in Konsument c_i wird die Kette um $(c_i, v'_{l|c_i|\tau_i})$ nach hinten erweitert.

Die Historie des initialen Tokens hat folgende Gestalt:

$$(\alpha, (d, 1)).$$

Dabei ist α ein Zeitpunkt vor dem ersten Füllschritt im Füllzyklus und d die Kontrollspalte des Cache-Keys. Da wir einen Füllzyklus für eine einzelnen Cache-Key-Wert betrachten, kommt dieser eine Wert beim Cache-Key an und provoziert einen Füllschritt.

Zu jedem Token existiert ein eindeutig bestimmtes Paar (τ, c) aus Erzeugungszeitpunkt τ und Erzeugungsspalte c. Das Token wurde ursprünglich zu diesem Erzeugungszeitpunkt in der Erzeugungsspalte generiert. Es können im allgemeinen aber mehrere Token mit dem gleichen Erzeugungszeitpunkt und der gleichen Erzeugungsspalte im SBN zirkulieren. Diese Token sind dann durch Duplikation (in Spalten mit mehreren ausgehenden RCCs) des gleichen ursprünglich generierten Tokens entstanden. Wir nennen das Paar (τ, c) die (Erzeugungs-)**ID** des Tokens.[27]

Die ID eines Tokens ist aus seiner Historie rekonstruierbar: Der Erzeugungszeitpunkt ist direkt in der Historie gespeichert und die Erzeugungsspalte ist die erste besuchte Spalte in der Besuchskette.

Beispiel: Wir wollen die Lebenszeit eines Tokens von der Generierung bis zum Verwerfen anschauen. Wir betrachten den WMP $c_1 \rightarrow \ldots \rightarrow c_n$, wobei c_1 ein Schmuggelziel sei. Zum Zeitpunkt τ_1 wird eine Menge von Werten $V'_{l|c_1|\tau_1}$ zusätzlich in c_1 geschmuggelt. Wegen dieser Erzeugung wird ein Token von c_1 generiert, das zunächst die Historie

$$(\tau_1, (c_1, v'_{l|c_1|\tau_1}))$$

speichert. Das Token befindet sich nach der Erzeugung in der Stelle, die zur RCC $c_1 \rightarrow c_2$ gehört. Vom Token-Manager wird dieses Token nun zum Zeitpunkt τ_2 aus der Stelle entnommen und in c_2 verarbeitet.

Die Verarbeitung des Tokens repräsentiert den Besuch zu einem Füllschritt, bei dem die Werte aus der Menge $V'_{l|c_1|\tau_1}$ in c_2 einstiegsfähig gemacht werden. Bei diesem Füllschritt ergibt sich die Wertemenge $V'_{l|c_2|\tau_2}$ der während des Füllschritts zusätzlich in c_2 induzierten Werte. Wir schätzen die Anzahl dieser Werte mit $v'_{l|c_2|\tau_2}$ ab[28] und verlängern die Besuchskette um den entsprechenden Eintrag.

---

[27] Obwohl die ID eigentlich die Erzeugung identifiziert und nicht das Token selbst (es kann durchschaus mehrere Token mit unterschiedlicher Historie aber der gleichen ID geben), sprechen wir der Einfachheit halber trotzdem von der ID des Tokens.

[28] Wir werden noch sehen, wie man diese Größe abschätzen kann, vorerst sei sie als gegebenen in den Raum gestellt.

Die Historie des Tokens sieht nun folgendermaßen aus:

$$(\tau_1, (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow (c_2,\ v'_{l|c_2|\tau_2}))$$

Nach der Verarbeitung wird das Token mit der verlängerten Historie in die Stelle zwischen c_2 und c_3 eingefügt. Zum Zeitpunkt τ_3 wird das Token in c_3 verarbeitet, die Besuchskette wieder verlängert und in die nächste Stelle gelegt. Schließlich kommt das Token irgendwann bei c_n an, wobei dann in ihm die Historie

$$(\tau_1, (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow (c_2,\ v'_{l|c_2|\tau_2}) \rightarrow \ldots \rightarrow (c_{n-1},\ v'_{l|c_{n-1}|\tau_{n-1}}))$$

gespeichert ist. Beim Füllschritt werden $v'_{l|c_n|\tau_n}$ zusätzlich in c_n induziert. Da diese Spalte aber keine ausgehenden RCCs hat, wird das Token schließlich verworfen.

5.2.2 Informationen aus der Historie

Nun wollen wir uns damit beschäftigen, welche Informationen wir aus einem ankommenden Token herauslesen können. Allgemein habe das ankommende Token die Historie

$$(\tau_1, (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow (c_2,\ v'_{l|c_2|\tau_2}) \rightarrow \ldots \rightarrow (c_{n-1},\ v'_{l|c_{n-1}|\tau_{n-1}})).$$

Wir gehen also davon aus, dass das Token zum Zeitpunkt τ_1 in der Spalte c_1 generiert wurde, und eine Reihe von Spalten auf einem WMP passiert hat, bevor es bei der betrachteten Spalte c_n ankommen ist. Bei der Erzeugung wurden die Werte aus einer Menge $V'_{l|c_1|\tau_1}$ zusätzlich in c_1 geladen (wobei wir nichts über die Zusammensetzung der Wertemenge sagen können; wir kennen auf Grund der Information im Token lediglich ihre Mächtigkeit).

Diese $v'_{l|c_1|\tau_1}$ Werte kamen zunächst bei c_2 an. Bei diesem Besuch wurden allerdings nicht alle Werte zusätzlich in c_2 induziert sondern nur ein Teil. Wir werden später (Abschnitt 5.4) noch sehen, aus welchen Gründen Werte verworfen werden, also nicht zu einem zusätzlichen Laden führen. Offensichtlich gilt aber:

$$V'_{l|c_1|\tau_1} \supseteq V'_{l|c_2|\tau_2}.$$

Allgemein können wir sagen, dass die besuchende Wertemenge eines Nachfolgers eine Teilmenge der besuchenden Wertemenge seines Vorgängers ist:

$$(c_i,\ v'_{l|c_i|\tau_i}) \rightarrow (c_{i+1},\ v'_{l|c_{i+1}|\tau_{i+1}}) \Rightarrow V'_{l|c_i|\tau_i} \supseteq V'_{l|c_{i+1}|\tau_{i+1}}. \qquad (5.2.2.1)$$

Aus der Historie des ankommenden Tokens können wir ablesen, wie viele Werte ankommen. Diese Größe ist im letzten Eintrag der Besuchskette in der Historie gespeichert. Daher wissen wir, dass $v'_{l|c_{n-1}|\tau_{n-1}}$ Werte beim betrachteten Füllschritt zum Zeitpunkt τ_n bei der Spalte c_n ankommen und durch den Füllschritt einstiegsfähig gemacht werden müssen. Diese Werte sind zuvor bei einem Füllschritt in die Spalte c_{n-1} geladen worden. Der Zeitpunkt dieses Füllschritts ist allerdings auf Grund der Historie nicht rekonstruierbar.

Da für Werte aus $V'_{l|c_n|\tau_n}$, die während des Füllschritts (zur Verarbeitung des Tokens gehörend) im Konsumenten c_n zusätzlich geladen wurden, ebenfalls diese Beziehung gilt, können

wir einen Ringschluss bilden, falls c_n ebenfalls die Erzeugungsspalte des Tokens ist, wenn also $c_1 = c_n$ gilt:

Zum Zeitpunkt τ_2 wird das Token in c_2 verarbeitet, was einem Besuch (c_2, $V'_{l|c_2|\tau_2}$) entspricht. Da c_2 zwei ausgehende RCCs hat, wird das Token entsprechend repliziert und in jede der nachfolgenden Stellen ein Duplikat eingefügt. Zum Zeitpunkt τ_3 zirkulieren also zwei Token mit der gleichen Historie im SBN.

$$V'_{l|c_n|\tau_1} \supseteq \ldots \supseteq V'_{l|c_{n-1}|\tau_{n-1}} \supseteq V'_{l|c_n|\tau_n}.$$

Da die Werte aber *zusätzlich* geladen sein müssen, ist die dargestellte Teilmengen-Beziehung nur möglich, wenn $V'_{l|c_n|\tau_n}$ leer ist. Eine solche Situation, dass ein Token bei seiner Erzeugungsspalte erneut ankommt, kann nur bei homogenen RCC-Zyklen auftreten. Denn wegen $c_1 = c_n$ muss das Token entlang eines WMP $c_n \rightarrow \ldots \rightarrow c_n$ gewandert sein. Dieser WMP ist ein homogener Zyklus.

Wird also von der verarbeitenden Spalte erkannt, dass sie die Erzeugungsspalte des ankommenden Tokens ist, so werden während des Füllschritts keine Werte zusätzlich geladen. Allerdings können durchaus Werte zusätzlich induziert werden. Die Werte wurden eventuell zum Erzeugungszeitpunkt in die Spalte geschmuggelt, sind aber nicht umgebungsvollständig. Dementsprechend können sie noch induziert werden und führen somit gegebenenfalls zu einem Laden von Sätzen aus dem Backend in die Cache-Tabelle. Das Token wird dennoch auf jeden Fall nach der Verarbeitung verworfen. Dadurch kann es durch homogene RCC-Zyklen nicht zu einem rekursiven Laden und damit exzessiven Füllverhalten kommen.

5.2.3 Besuchsbäume

Wir können die Besuche aus Historien von Token, die ursprünglich bei der gleichen Erzeugung generiert und in Spalten mit mehreren ausgehenden RCCs repliziert wurden in **Besuchsbäume** zusammenfassen. Diese Token haben dann die gleiche ID und den gleichen ersten Besuch in der Besuchskette.

Die Knoten dieser Bäume sind Besuche und die Kanten entsprechen der Vorgänger-Nachfolger-Beziehung in einer Besuchskette der Token. Die Wurzel des Baums ist die Erzeugung der Token und der Baum ist genau wie die zugehörigen ursprünglichen Token eindeutig anhand des Erzeugungszeitpunkts und der Erzeugungsspalte zu identifizieren.

Wir werden die Besuchsbäume zur Erklärung der Kontext-Bäume (Abschnitt 5.3) benötigen, die wiederum wichtige Datenstrukturen zur Speicherung der Informationen über bereits angekommene Wertemengen darstellen. Aber auch so lassen sich interessante Informationen aus den Besuchsbäumen ableiten.

Schauen wir uns als Beispiel die Token aus Abbildung 20 an. Zum Zeitpunkt τ_9 zirkulieren vier Token im SBN. Alle Token wurden jedoch zum Zeitpunkt τ_1 bei einem Besuch (c_1, $V'_{l|c_1|\tau_1}$) generiert.

Der Token-Manager wählt nun (willkürlich) das Token vor c_3 aus und lässt einen Füllschritt durch die Cache-Tabelle von c_3 ausführen. Die Spalte c_3 wird nun aktiv und bei dem Besuch

(c_3, $V'_{l|c_3|\tau_3}$) wird das Token verarbeitet. Dabei wird die Besuchskette verlängert und das Token weiter gegeben.

Zum Zeitpunkt τ_4 wird die Spalte c_4 aktiv und verarbeitet das bei ihr ankommende Token. Auch hier wird durch die Bearbeitung die Besuchskette in der Historie des Tokens verlängert und das Token weiter gegeben.

Danach kommen beide Token bei c_5 an. Wir schauen uns die Historie der beiden Token zum Zeitpunkt τ_5 an. Die Historien haben folgende Gestalt:

Token 1) $(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow (c_2, v'_{l|c_2|\tau_2}) \rightarrow (c_3, v'_{l|c_3|\tau_3}))$

Token 2) $(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow (c_2, v'_{l|c_2|\tau_2}) \rightarrow (c_4, v'_{l|c_4|\tau_4}))$

Die Historien der beiden Token lassen sich zu einem Beziehungsbaum zusammenfassen, der zum Zeitpunkt τ_5 wie in Abbildung 21 dargestellt aussieht:

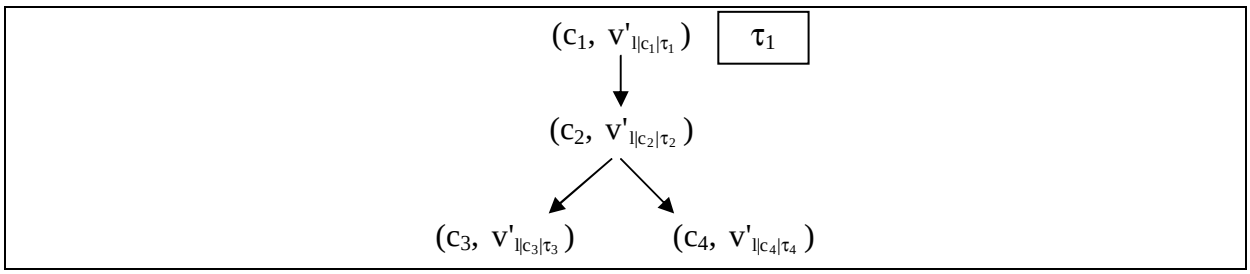

Abbildung 21: Besuchsbaum

Die Struktur ergibt tatsächlich immer einen Baum, denn selbst wenn eine Spalte in zwei unterschiedlichen Historien vorkommt, so wird kein gerichteter, azyklischer Graph gebildet. Als Beispiel schauen wir uns Zeitpunkt τ_8 an. Zuvor wurden die beiden Token nacheinander in der Spalte c_5 verarbeitet und die Historie jeweils verlängert:

Token 1) $(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow (c_2, v'_{l|c_2|\tau_2}) \rightarrow (c_3, v'_{l|c_3|\tau_3}) \rightarrow (c_5, v'_{l|c_5|\tau_6}))$

Token 2) $(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow (c_2, v'_{l|c_2|\tau_2}) \rightarrow (c_4, v'_{l|c_4|\tau_4}) \rightarrow (c_5, v'_{l|c_5|\tau_7}))$

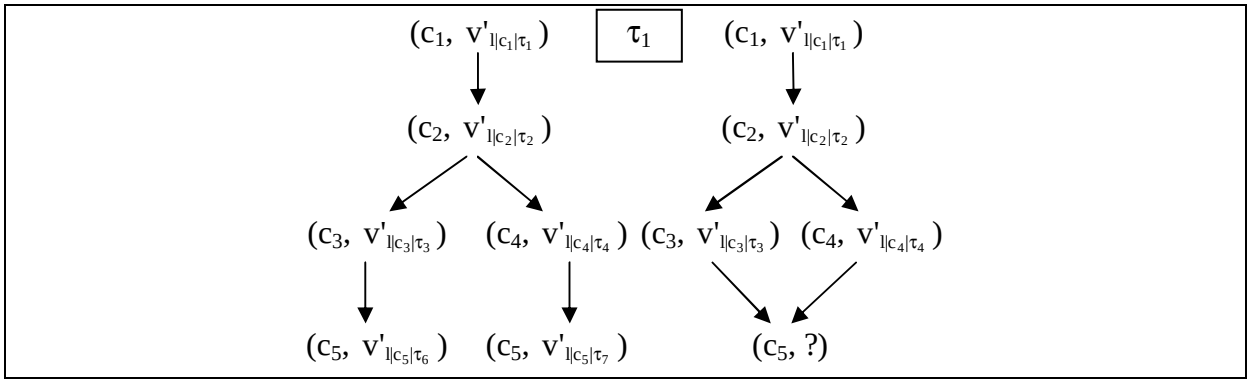

Abbildung 22: Besuchsbaum und gerichteter, azyklischer Graph mit Besuchen

Abbildung 22 zeigt auf der linken Seite den Besuchsbaum, der aus den Besuchsketten der beiden Historien entsteht. Auf der rechten Seite ist ein gerichteter, azyklischer Graph abgebildet, der dadurch entstanden ist, dass nacheinander besuchte Spalten durch Kanten miteinander

verbunden wurden. Allerdings ist in diesem Fall unklar, wie groß die Anzahl der besuchenden Werte ist, die in der Spalte c_5 ankamen. Dies ist durch ein Fragezeichen gekennzeichnet.

5.3 Kontexte und Kontext-Bäume

Betrachten wir noch einmal einen bestimmten Konsumenten k, der im Laufe des Füllzyklus mehrmals aktiv wird und Token verarbeitet. Die gesammelten Informationen über die Besuche, die ankommende Token gemacht haben, bevor sie von k verarbeitet wurden, werden dort verwaltet.

Die Information aus den Besuchen steht in so genannten **Kontexten** zur Verfügung. Mit Hilfe von Kontexten können wir entscheiden, wie viele der ankommenden Werte, die einen gemeinsamen Besuch haben, bereits angekommen sind. Analog zu Besuchen unterscheiden wir zwischen manifestierten und abstrahierten Kontexten, wobei wir die manifestierten Kontexte wieder nur zur Erläuterung der Bedeutung benötigen.

Ein abstrahierter Kontext $\hbar$ ist ein Tripel $(c, v'_{l|c|\tau}, v_\hbar)$. Wir sagen, der Kontext **gehört zum** Besuch $(c, v'_{l|c|\tau})$ und nennen $v_\hbar$ den **Füllstand** des Kontextes $\hbar$. Der Füllstand gibt die Anzahl der Werte an, die beim zugehörigen Besuch in der besuchten Spalte c zusätzlich geladen *und* im Konsumenten k *induziert* sind.

Ein Kontext erweitert somit gewissermaßen das Wissen über den Besuch aus der Sicht des Konsumenten k. Er gibt uns zusätzliche Informationen über die Werte aus dem Füllstand; aus dem Kontext können wir ablesen, bei welchen Spalten die besuchenden Wertemengen angekommen sind. Die Werte gehören also zu diesem Umfeld bzw. eben Kontext, womit wir eine Erklärung für den Namen haben.

Ein Kontext ist abhängig vom betrachteten Zeitpunkt, d. h. der Füllstand ändert sich im Laufe des Füllzyklus. Da wir Kontexte aber immer zu einem bestimmten Zeitpunkt betrachten (nämlich immer zum Zeitpunkt der Verarbeitung eines Tokens), können wir uns die Angabe des Index zur Bestimmung des Zeitpunkts im Allgemeinen sparen.

Solch ein abstrahierter Kontext wird natürlich zur Modellierung eines manifestierten Kontextes eingesetzt. Ein manifestierter Kontext ist ein Tripel $(c, V'_{l|c|\tau}, V_\hbar)$, wobei wir in diesem Fall $V_\hbar$ die Menge der **Kontext-Füllwerte** nennen.

Die Menge der Kontext-Füllwerte ist immer eine Teilmenge der besuchenden Wertemenge:

$$V_\hbar \subseteq V'_{l|c|\tau} \text{ für } (c, V'_{l|c|\tau}, V_\hbar) \qquad (5.3.1)$$

Genau wie bei den Besuchsbäumen können wir Kontexte in **Kontext-Bäumen** anordnen. Die Knoten eines Kontext-Baums sind dabei Kontexte und zwischen zwei Kontexten ist genau dann eine Kante, wenn zwischen den zugehörigen Besuchen im entsprechenden Besuchsbaum auch eine Kante ist.

Genau wie ein Beziehungsbaum ist ein Kontext-Baum genau einem Erzeugungszeitpunkt und genau einer Erzeugungsspalte zugeordnet. Wir verwalten in jeder Spalte maximal einen Kontext-Baum pro ID der Token. Ist noch kein Token mit einer bestimmten ID beim Konsumenten angekommen, so existiert dort auch noch kein entsprechender Kontext-Baum. Erst

nach der ersten Verarbeitung eines Tokens mit dieser ID wird ein entsprechender Kontext-Baum angelegt.

Beispiel: Wir wollen die Entstehung eines Kontext-Baums wieder anhand eines Beispiels betrachten. Wir schauen uns dafür die Abbildung 20 an und nehmen die Zeitpunkte τ_6 und τ_7 unter die Lupe. Dabei betrachten wir, wie der Kontext-Baum zu den ankommenden Token in der Spalte c_5 vor und nach der Verarbeitung aussieht.

Natürlich ist der Kontext-Baum vor der ersten Ankunft eines entsprechenden Tokens leer. Wir betrachten also zunächst den Füllschritt zum Zeitpunkt τ_6. Das ankommende Token hat eine Historie mit folgender Gestalt:

$$(\tau_1, (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow (c_2,\ v'_{l|c_2|\tau_2}) \rightarrow (c_3,\ v'_{l|c_3|\tau_3}))$$

In c_5 werden während des Füllschritts $v'_{l|c_5|\tau_6}$ Werte zusätzlich geladen. Ohne konkret zu wissen, wie diese Größe zustande kommt, wollen wir sie zunächst als gegeben hinnehmen und sehen, wie sich dies auf die Füllstände der verschiedenen Kontexte auswirkt. Die zusätzlich geladenen Werte stammen aus der Erzeugung in Spalte c_1 und kamen der Reihe nach bei c_2 und c_3 an. Dabei wurde jeweils eine Reihe von Werten verworfen. Die nun zusätzlich geladenen Werte liegen also alle sowohl in $V'_{l|c_1|\tau_1}$, $V'_{l|c_2|\tau_2}$ als auch in $V'_{l|c_3|\tau_3}$. Sie liegen in den besuchenden Wertemengen und sind in c_5 durch Verarbeitung eines entsprechenden Tokens (nämlich des betrachteten) geladen worden. Für den Kontext-Baum ergibt sich somit die in Abbildung 23 links dargestellte Gestalt. Die Füllstände in allen Kontexten sind gleich: $v'_{l|c_5|\tau_6}$.

Zum Zeitpunkt τ_7 kommt das zweite Token bei c_5 an. Das Token hat die Historie

$$(\tau_1, (c_1,\ v'_{l|c_1|\tau_1}) \rightarrow (c_2,\ v'_{l|c_2|\tau_2}) \rightarrow (c_4,\ v'_{l|c_4|\tau_4})).$$

Bei diesem Füllschritt werden in die Spalte $v'_{l|c_5|\tau_7}$ Werte zusätzlich geladen.

Wir wollen nun sehen, wie der Kontext-Baum verändert werden muss. Zum Besuch $(c_4, v'_{l|c_4|\tau_4})$ existiert noch kein Pendant im Kontext-Baum, also müssen wir einen solchen Kontext einfügen. Der Füllstand dieses Kontextes ist die Anzahl der im aktuellen Füllschritt zusätzlich geladenen Werte, denn diese bilden eine Teilmenge der die Spalte c_4 besuchenden Werte.

Die Füllstände der anderen Kontexte, die zu einem Besuch in der Historie des Tokens gehören, müssen natürlich ebenfalls erhöht werden. Die Werte, die vorher in der entsprechenden Menge lagen, liegen natürlich immer noch darin. Die während des aktuellen Füllschritts zusätzlich geladenen Werte erweitern allerdings die Menge der Kontext-Füllwerte, kommen also hinzu. Der neue Füllstand ergibt sich also jeweils aus der Summe des alten Füllstands zuzüglich der zusätzlich geladenen Werte.

Nach dem Füllschritt sieht der Kontext-Baum so aus, wie er in Abbildung 23 auf der rechten Seite abgebildet ist.

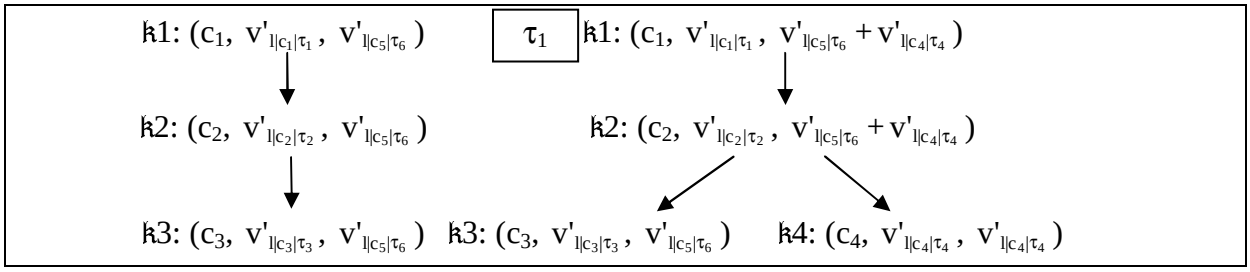

Abbildung 23: Kontext-Bäume in Spalte c_5 zum Zeitpunkt τ_6 (links) bzw. τ_7 (rechts)

Wir wollen nun allgemein betrachten, wie neue Informationen aus der Historie ankommender Token in Kontext-Bäume übernommen werden können. Wir bezeichnen diesen Vorgang der Aktualisierung eines Kontext-Baums als die **Integration** des Tokens.

Wir müssen bei der Verarbeitung eines ankommenden Tokens in der verarbeitenden Spalte k zum Zeitpunkt τ zwei Situationen unterscheiden:

- Existiert zur ID des ankommenden Tokens (also zu seinem Erzeugungszeitpunkt und seiner Erzeugungsspalte) noch kein Kontext-Baum in der verarbeitenden Spalte k, so entsteht ein neuer Kontext-Baum aus der Historie des ankommenden Tokens. Die Kontexte entsprechen dabei den Besuchen der Besuchskette und werden um die Anzahl $v'_{l|k|\tau}$ der während des Füllschritts zusätzlich geladenen Werte als Füllstand erweitert. Dieser Füllstand ist in jedem Kontext des Kontext-Baums gleich.

- Wird andererseits bereits ein Kontext-Baum zur ID des Tokens verwaltet, so müssen nur die zusätzlichen Besuche in den Kontext-Baum eingefügt werden. Wir können dazu die Besuchskette des Tokens in zwei Teile zerlegen. Der erste Teil der Besuchskette ist bereits im verwalteten Kontext-Baum vorhanden. Für den ersten Besuch der Historie gilt dies auf jeden Fall, denn ansonsten gäbe es noch keinen Kontext-Baum für die ID des Tokens.

  Wir können uns entlang der Besuchskette hangeln (d. h. wir betrachten die Besuche in der Besuchskette iterativ und beginnen bei der Erzeugung) und testen, ob der betrachtete Besuch jeweils einem Kontext im Kontext-Baum entspricht. Dabei muss ebenfalls eine Kante vom Vorgänger aus zum gesuchten Kontext existieren. Wird zum betrachteten Besuch kein Kontext gefunden, so handelt es sich um einen **neuen** Kontext. Alle nachfolgenden Besuche sind dann ebenfalls neu. Um die Kontexte unterscheiden zu können, sprechen wir von den Kontexten, die bereits im Kontext-Baum waren, als **alte** Kontexte.

  In den alten Kontexten muss nun der Füllstand erhöht werden. Es kommen diejenigen Werte hinzu, die während des Füllschritts zusätzlich geladen wurden. In den neuen Kontexten entspricht der Füllstand der Anzahl $v'_{l|k|\tau}$ der zusätzlich geladenen Werte.

  In unserem Beispiel wären die Kontexte ħ1 und ħ2 alt und der Kontext ħ4 neu, wenn wir den Füllschritt zum Zeitpunkt τ_7 in der verarbeitenden Spalte c_5 betrachten.

5.4 Abschätzung der Anzahl zusätzlich geladener Werte

In den vorangegangenen Abschnitten haben wir gesehen, wie Token den „Fluss“ von Werten in einer Cache-Group oder -Föderation modellieren. Wir haben außerdem die Semantik der Token betrachtet, nämlich dass sie in unserem Modell Wertemengen repräsentieren. Die Token speichern Informationen, die sie auf ihrem Weg über WMPe gesammelt haben, in ihrer Historie. Die Historie wiederum besteht aus dem Erzeugungszeitpunkt und einer Besuchskette. Wir haben des Weiteren betrachtet, wie man die Historien mehrerer Token mit gleichem Erzeugungszeitpunkt und Erzeugungsspalte in Besuchsbäumen zusammenfassen kann, und wie diese durch Erweiterung mit Informationen über die Kontext-Füllwerte zu Kontext-Bäumen werden.

Eine große Rolle bei der Verarbeitung der Token und bei der Bildung von Kontext-Bäumen spielen dabei die Anzahl der jeweils zusätzlich geladenen Werte. Diese Größe haben wir bisher als gegeben angesehen und noch nicht erörtert, wie man sie abschätzen kann.

In diesem Abschnitt wollen wir uns nun der Frage widmen, wie man die Anzahl während eines Füllschritts zusätzlich geladener Werte berechnen kann. Wir betrachten dazu wieder den Zeitpunkt τ, bei dem ein gegebener Konsument k (:= c_n) aktiv ist und ein Token mit der Historie

$$(\tau_1, (c_1, v'_{|c_1|\tau_1}) \rightarrow \ldots \rightarrow (c_{n-1}, v'_{|c_{n-1}|\tau_{n-1}}))$$

verarbeitet. Wie üblich nehmen wir also ohne Beschränkung der Allgemeinheit an, dass das ankommende Token zuvor über einen WMP $c_1 \rightarrow \ldots \rightarrow c_{n-1}$ gewandert ist.

Wir können zwei Einflussgrößen identifizieren, die dazu führen, dass ein ankommender Wert während des Füllschritts nicht zusätzlich geladen wird:

(1) Entweder der ankommende Wert liegt außerhalb der DBD der verarbeitenden Spalte k oder

(2) der ankommende Wert war vorher bereits in k geladen.

Man beachte, dass die Eigenschaften (nicht in der DBD zu liegen bzw. bereits geladen zu sein) nicht unabhängig sind, sondern sich vielmehr widersprechen. Ein Wert, der nicht in der DBD einer Spalte liegt, kann natürlich nicht zuvor geladen worden sein und umgekehrt. Wir nennen die ankommenden Werte, die in der DBD der verarbeitenden Spalte liegen, **generell zu laden** und schreiben für die Menge dieser Werte $V'_{g|k|\tau}$

Die Werte, welche einen der oben genannten Punkte (1) oder (2) erfüllen, führen nicht dazu, dass eine RCC zu eventuellen Nachfolgern nicht mehr gilt. Dennoch kann ein ankommender Wert, der vorher bereits geladen war, zu einer Erhöhung des Füllstands der Cache-Tabelle führen. Dies ist immer dann der Fall, wenn der Wert zuvor zwar bereits geladen, aber noch nicht umgebungsvollständig war. In diesem Fall muss noch ein Teil der Werteumgebung in die Cache-Tabelle geladen werden, um den Wert einstiegsfähig zu machen. Das wiederum kann allerdings ausgehende RCCs an den Schmuggelzielen der Cache-Tabelle ungültig machen und so weitere Füllschritte provozieren.

Für einen ankommenden, vorher bereits geladenen Wert gibt es wiederum zwei Möglichkeiten:

- Der Wert ist vor dem Füllschritt umgebungsfragmentarisch in k, dann müssen noch Sätze aus dem Backend geholt werden, damit die RCC wieder gelten kann.
- Der Wert ist bereits umgebungsvollständig in k, dann führt er nicht zu einer Erhöhung des Füllstands $n_{T(k)}$ der Cache-Tabelle T(k).

In jedem Fall führt ein ankommender, bereits geladener Wert nicht zu einer Nicht-Erfüllung der RCC-Eigenschaft. Der Wert „fließt“ also nicht weiter über ausgehende RCCs. Insofern werden im – durch den Füllschritt entstehenden – abstrahierten Besuch auch nur die tatsächlich zusätzlich geladenen Werte aufgeführt.

5.4.1 Anzahl generell zu ladender Werte

Wir wollen nun zunächst betrachten, wie viele der ankommenden Werte innerhalb der DBD liegen, also generell zu laden sind. Wir wissen, dass die Werte in c_{n-1} geladen sind und in der Schnittmenge der DBDs von c_{n-1} und k liegen. Als eine gemeinsame Obermenge können wir die DBD von c_{n-1} identifizieren.

Für eine Average-Case-Betrachtung wenden wir Formel 2.5.1 an und erhalten:

$$v'_{g|k|\tau} \approx v'_{l|c_{n-1}|\tau_{n-1}} \cdot |V_{c_{n-1}} \cap V_k| / c_{n-1} = v'_{l|c_{n-1}|\tau_{n-1}} \cdot \ddot{u}_{k,c_{n-1}}.$$

Da wir aber von einer Worst-Case-Analyse ausgehen, nehmen wir an, dass so viele ankommende Werte wie möglich in der DBD von k liegen. Wenn die Anzahl der ankommenden Werte größer als die Schnittmenge der DBDs von c_{n-1} und k ist, so kann dies natürlich nur für einen Teil der Werte gelten; der Rest muss außerhalb der DBD von k liegen.

Die Anzahl $v'_{g|k|\tau.}$ der generell zu ladenden Werte wird außerdem um die Anzahl $v_{i|c_{n-1}\rightarrow k|\tau}$ der Werte reduziert, die bereits über c_{n-1} angekommen sind. Ansonsten gibt es einen Widerspruch zur Annahme, dass die ankommenden Werte im Vorgänger zusätzlich geladen wurden.

Wir können nun also die Anzahl der ankommenden und in der DBD des Konsumenten liegenden Werte im Sinne einer Worst-Case-Analyse wie folgt abschätzen:

$$v'_{g|k|\tau} \leq \min(v'_{l|c_{n-1}|\tau_{n-1}}, |V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1}\rightarrow k|\tau}).$$

Man beachte, dass die dargestellte obere Schranke

$$|V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1}\rightarrow k|\tau}$$

in jedem Fall größer als null ist, denn die über diese RCC indizierten Werte müssen ebenfalls in der Schnittmenge der DBDs liegen.

Wir können die Anzahl $v_{i|c_{n-1}\rightarrow k|\tau}$ aus den Kontext-Füllständen rekonstruieren. Wir betrachten dazu jeden Kontext, der ein Blatt in einem Kontext-Baum ist und dessen besuchte Spalte c_{n-1} ist. Wir aktualisieren dann die Füllstände der qualifizierten Kontexte. Dies funktioniert, da die besuchten Spalten in den Blättern der Kontext-Bäume die Vorgänger der verarbeitenden Spalte sind. Insbesondere ist dann auch c_{n-1} darunter zu finden. Die Kontext-Füllstände geben die jeweilige Anzahl der Werte an, die bei der besuchten Spalte c_{n-1} und bei der verarbeitenden Spalte k angekommen sind. Dementsprechend wurden sie über die RCC $c_{n-1}\rightarrow k$ induziert.

Bei der Berechnung der Anzahl generell zu ladender Werte gibt es einen Spezialfall: die Verarbeitung des initialen Tokens im Cache-Key. Da wir einen Füllzyklus nach einem Cache-Miss betrachten und dabei davon ausgehen, dass der referenzierte Cache-Key-Wert in der DBD des Cache-Keys liegt (also tatsächlich ein Füllzyklus provoziert wird), ist die DBD der Kontrollspalte gleich der DBD des Cache-Keys. Die Anzahl der generell zu ladenden Werte im ersten Füllschritt ist also immer gleich 1.

5.4.2 Anzahl der zusätzlich geladenen Werte

Wir kennen nun die Anzahl der generell zu ladenden Werte und wollen im Rest des Abschnitts betrachten, wie man die Anzahl der zusätzlich geladenen Werte abschätzen kann. Diese Größe wird wiederum durch die Anzahl der ankommenden, bereits geladenen Werte reduziert. Wir unterscheiden wieder drei Fälle:

- Der Wert wurde bereits induziert, d. h. ist mit einem Token angekommen. In diesem Fall ist der Wert umgebungsvollständig und wird nicht zusätzlich geladen.
- Der Wert wurde zuvor in die verarbeitende Spalte geschmuggelt. Dann kann der Wert sowohl umgebungsvollständig als auch umgebungsfragmentarisch sein.
- Der Wert wurde weder induziert, noch in die Spalte geschmuggelt. Dann ist der Wert ungeladen und es muss beim Füllschritt noch seine vollständige Werteumgebung in die Cache-Tabelle geladen werden.

Die letzten beiden Fälle führen dazu, dass der ankommende Wert zusätzlich geladen wird. Im zweiten Fall wird der Wert zusätzlich induziert und führt damit zu einer Erhöhung des Füllstands, wenn der Wert noch nicht umgebungsvollständig war. Im letzten Fall wird der Wert unter allen Umständen zusätzlich induziert und der Füllstand erhöht sich um die Mächtigkeit der Werteumgebung des Werts.

Wir können die Anzahl der Werte, von denen wir definitiv wissen, dass sie bereits induziert sind, von den gespeicherten Kontext-Bäumen ablesen. Wenn wir einen Kontext-Baum zur ID des Tokens finden, so wissen wir, dass bereits einige Werte in k induziert wurden, die ursprünglich (zum Erzeugungszeitpunkt) in die Erzeugungsspalte eingeführt worden sind. Gleiches können wir eventuell auch von späteren Besuchen in der Besuchskette des Tokens sagen.

Wir können in analoger Weise wie bei der Integration des Tokens wieder die Besuchskette des ankommenden Tokens in zwei Teile zerlegen. Der erste Teil der Kette entspricht alten Kontexten, die bereits im Kontext-Baum der ID gespeichert sind. Wir finden diese Kontexte, indem wir uns entlang der Besuchskette hangeln und im Kontext-Baum den gleichen Weg nachvollziehen; zwischen zwei Besuchen, die aufeinander folgen, muss im Kontext-Baum eine Kante zwischen den zugehörigen Kontexten sein. Wir verfolgen die Besuchskette und durchlaufen den Pfad im Kontext-Baum von der Wurzel zu den Blättern so lange, bis wir zu einem Besuch in der Kette kommen, der noch nicht im Kontext-Baum ist. Der letzte Besuch in der Besuchskette, zu dem wir einen Kontext im Kontext-Baum finden, nennen wir **letzten gemeinsamen Besuch**. Analog nennen wir den zugehörigen Kontext **letzter gemeinsamer Kontext**. Indem wir die Besuchskette hinter dem letzten gemeinsamen Besuch aufteilen,

können wir sie in einer Kette von **alten** Besuchen (vor dem Schnitt) und **neuen** Besuchen teilen.

Von den besuchenden Wertemengen der alten Besuche kennen wir die Anzahl der bereits in k induzierten Werte (aus dem Kontext-Füllstand). Die Differenzen zwischen der jeweiligen Mächtigkeit der besuchenden Wertemenge und dem zugehörigen Kontext-Füllstand stellen obere Schranken für die Anzahl der zusätzlich zu induzierenden Werte dar. Die Kontext-Füllwerte $V_ħ$ liegen alle in der besuchenden Wertemenge und wurden bereits induziert, müssen also nicht zusätzlich geladen werden.

Beispiel: Nehmen wir an, wir finden einen Kontext-Baum für die ID (c_1, τ_1) des ankommenden Tokens. Der Kontext-Baum wird nun von der Wurzel in Richtung der Blätter entsprechend der Besuche durchlaufen, indem wir die Kanten zum Kontext mit der jeweilig nächsten besuchten Spalte verfolgen. Wir finden dabei die alten Kontexte ħ1 bis ħi. Für den Besuch (c_{i+1}, $v'_{l|c_{i+1}|\tau_{i+1}}$) wird kein entsprechender Kontext im Kontext-Baum gefunden, d. h. es gibt keinen Sohn (c_{i+1}, $v'_{l|c_{i+1}|\tau_{i+1}}$, $v_ħ$) zum letzten gemeinsamen Kontext (c_i, $v'_{l|c_i|\tau_i}$, $v_{ħi}$).

Aus der Differenz zwischen der Mächtigkeit der besuchenden Wertemenge und dem Kontext-Füllstand lassen sich i obere Schranken für die Anzahl der zusätzlich zu ladenden Werte bilden:

$$v'_{l|k|\tau} \leq v'_{l|c_1|\tau_1} - v_{ħ1}, \ldots, v'_{l|k|\tau} \leq v'_{l|c_i|\tau_i} - v_{ħi}$$

Eine weitere obere Schranke stellt die Anzahl der ungeladenen Werte dar. Natürlich können maximal nur so viele Werte beim Füllschritt zusätzlich geladen werden, wie vorher ungeladen waren:

$$v'_{l|k|\tau} \leq v_{u|k|\tau}$$

Für eine Worst-Case-Analyse nehmen wir nun an, dass die maximal mögliche Anzahl von Werten während des Füllschritts zusätzlich geladen wird. Die Anzahl der zusätzlich geladenen Werte wird also durch die kleinste obere Schranke angenähert:

$$v'_{l|k|\tau} = \min(v'_{l|c_1|\tau_1} - v_{ħ1}, \ldots, v'_{l|c_i|\tau_i} - v_{ħi}, v_{u|k|\tau}, v'_{g|k|\tau})$$

$$= \min(v'_{l|c_1|\tau_1} - v_{ħ1}, \ldots, v'_{l|c_i|\tau_i} - v_{ħi}, v_{u|k|\tau}, |V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1}\to k|\tau}, v'_{l|c_{n-1}|\tau_{n-1}}).$$

Analog können wir die Anzahl der zusätzlich induzierten Werte berechnen. Hier verwenden wir allerdings nicht die Anzahl der ungeladenen Werte als obere Schranke, sondern die Anzahl der ungeladenen zuzüglich der Anzahl der fragmentarischen Werte:

$$v_{i|k|\tau} = \min(v'_{l|c_1|\tau_1} - v_{ħ1}, \ldots, v'_{l|c_i|\tau_i} - v_{ħi}, v_{u|k|\tau} + v_{vf|k|\tau}, |V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1}\to k|\tau}, v'_{l|c_{n-1}|\tau_{n-1}}).$$

Ein Sonderfall entsteht dann, wenn das ankommende Token im Konsumenten k erzeugt wurde. Wir können diese Situation anhand der Historie des Tokens identifizieren. In diesem Fall gilt nämlich $c_1 = k$. Das Token ist dann über einen homogenen Zyklus gewandert und führt dementsprechend nicht zu einem zusätzlichen Laden von Werten. Es gilt also in diesem Fall:

$$v'_{l|k|\tau} = 0.$$

Dennoch können einige der ankommenden Werte zusätzlich induziert werden. Die Werte wurden bei der Erzeugung in k eingeführt und sind deswegen auf jeden Fall geladen. Bei Nicht-Unique-Spalten kann es jedoch sein, dass die Werte umgebungsfragmentarisch waren. Wegen der Worst-Case-Betrachtung gehen wir von diesem Fall aus, denn so wird die maximale Anzahl der zusätzlich induzierten Werte angenommen. Natürlich wird die Anzahl durch die Anzahl der umgebungsfragmentarischen Werte beschränkt:

$$v_{i|k|\tau} = \min(v_{vf|k|\tau},\ v'_{l|c_{n-1}|\tau_{n-1}}).$$

Da bei Unique-Spalten keine umgebungsfragmentarischen Werte vorkommen, werden auch keine Werte zusätzlich induziert.

Ansatzpunkte für eine Average-Case-Analyse

Bei einer Average-Case-Betrachtung müssten wir die Größe der Schnittmenge zwischen den generell zu ladenden und den aus anderen Erzeugungen induzierten sowie den geschmuggelten Werten bilden, um die Anzahl der zusätzlich geladenen Werte zu berechnen. Für diese Berechnung müssten wir allerdings betrachten, von welchen Werten wir definitiv wissen, dass sie nicht in der Schnittmenge liegen, d. h. dass sie nicht ankommen. Hierfür gibt es mehrere Beispiele:

- Die Werte aus der Erzeugungsspalte, die bereits angekommen sind, können nicht erneut ankommen, denn ansonsten wären sie dort mehr als einmal zusätzlich geladen. Insbesondere gilt dies für die Füllwerte der Kontexte, die die Wurzel eines zur Erzeugungsspalte gehörenden Kontext-Baums sind.
- Gleiches gilt für die Werte, die bereits über den gleichen Vorgänger induziert wurden. Auch hier können diese Werte nicht erneut ankommen.

Wir können dieses Prinzip verallgemeinern, indem wir feststellen, dass ankommende Werte disjunkt zu den Kontext-Füllwerten sind, die zu einem Kontext-Baum mit anderer Erzeugungsspalte und/oder Erzeugungszeitpunkt gehören. Wir können die Kontext-Füllstände aber nicht einfach suchen und addieren, denn die zugehörigen Wertemengen können sich ebenfalls überschneiden; deswegen müssten wir wieder Schnittmengenmächtigkeiten berechnen. Alles in allem würde das Modell hierdurch so komplex, dass der Nutzen (im Sinne der Laufzeit des Algorithmus) die Kosten nicht rechtfertigen könnte.

5.5 Veränderung des Füllstands

Im vorangegangenen Teil des Kapitels haben wir unter anderem gesehen, wie Token verarbeitet werden und wie die Anzahl zusätzlich induzierter bzw. geladener Werte berechnet werden kann. Diese Werte haben natürlich eine Auswirkung auf den Füllstand der Cache-Tabelle. Wir unterscheiden dabei folgende Arten von Werten:

- Werte, die zwar zusätzlich induziert aber nicht zusätzlich geladen werden, waren vor dem Füllschritt wertfragmentarisch. Für diese Werte muss nicht mehr die komplette Werteumgebung in den Cache geholt werden, sondern nur ein Teil davon.

- Die restlichen Werte wurden zusätzlich geladen und für sie muss die komplette Werteumgebung in den Cache geholt werden.

Letztere Werte führen natürlich zu einem größeren Anwachsen des Füllstands der Cache-Tabelle, da für sie mehr Sätze aus dem Backend geladen werden. Wir können die Anzahl $s'_{l|k|\tau}$ der Sätze, die für diese Werte in die Cache-Tabelle geladen werden, wie folgt berechnen, wenn wir von der Gleichverteilungsannahme ausgehen:

$$s'_{l|k|\tau} = v'_{l|k|\tau} \cdot |S_{T(k)}| / c_k.$$

Die Werte, die vor dem Füllschritt bereits wertfragmentarisch waren, führen zu einem geringeren Anwachsen des Füllstands. Hier muss die Anzahl der Sätze abgezogen werden, die in der Werteumgebung der zusätzlich induzierten Werte liegen und in der Cache-Tabelle geladen sind. Insgesamt ergibt sich für die Anzahl $s'_{T(k)|\tau}$ der in die Cache-Tabelle während des Füllschritts zusätzlich geladenen Sätze:

$$s'_{T(k)|\tau} = s'_{i|k|\tau} - |S'_{l|k|\tau} \cap S_{l|T|\tau}|.$$

Dabei ist $s'_{i|k|\tau}$ die abgeschätzte Mächtigkeit der Werteumgebung der zusätzlich induzierten Werte, $S'_{l|k|\tau}$ ist die Werteumgebung der zusätzlich geladenen Werte und $S_{T|\tau}$ ist die Menge der Sätze, die zum Zeitpunkt τ in der Cache-Tabelle T(k) geladen sind.

Für die Suche nach einer oberen Schranke für $s'_{T(k)|\tau}$ können wir eine untere Schranke für die Mächtigkeit der in Abzug zu bringenden Schnittmengenmächtigkeit benutzen. Ein naiver Ansatz wäre es, anzunehmen, dass die Schnittmenge leer ist. Dies würde nämlich der Tatsache widersprechen, dass die zusätzlich induzierten aber nicht zusätzlich geladenen Werte vorher umgebungsfragmentarisch waren.

Natürlich muss zu jedem umgebungsfragmentarischen Wert mindestens ein Satz aus seiner Werteumgebung in der Cache-Tabelle geladen sein. Insbesondere gilt dies natürlich für die Sätze, die zwar zusätzlich induziert, aber nicht zusätzlich geladen wurden:

$$|S'_{l|k|\tau} \cap S_{l|T|\tau}| \geq v'_{i|k|\tau} - v'_{l|k|\tau}.$$

Auf der anderen Seite muss natürlich jeder zusätzlich geladene Satz im Backend vorhanden sein. Die Anzahl der während des Füllschritts in die Cache-Tabelle geladenen Sätze kann dementsprechend niemals die Anzahl der Sätze übersteigen, die noch nicht in der Cache-Tabelle sind:

$$s'_{T(k)|\tau} \leq |S_{T(k)}| - s_{l|T(k)|\tau}$$

Auf Grund der Worst-Case-Betrachtung machen wir aus den Ungleichungen wieder eine Minimum-Funktion:

$$s'_{T(k)|\tau} = \min(s'_{i|k|\tau} - v'_{i|k|\tau} + v'_{l|k|\tau}, |S_{T(k)}| - s_{l|T(k)|\tau}).$$

5.6 Aktualisierung der Statistik im Konsumenten

Nachdem wir nun betrachtet haben, wie wir die Anzahl der zusätzlich geladenen und zusätzlich induzierten Werte und der im Füllschritt geladenen Sätze berechnen können, wollen wir

uns kurz anschauen, wie wir die Statistik im Konsumenten nach dem Füllschritt aktualisieren. Dazu definieren wir die Schreibweise $\tau+$ für den Zeitpunkt unmittelbar nach dem Füllschritt.

Zunächst müssen wir das angekommene Token in den verwalteten Kontext-Baum integrieren. Wurde noch kein Kontext-Baum mit der entsprechenden ID gefunden, so wird ein neuer Kontext-Baum aus der Historie des Tokens erzeugt. Für jede ID existiert also in jeder Spalte maximal ein Kontext-Baum. Wie die Integration des Tokens funktioniert, haben wir bereits in Abschnitt 5.3 gesehen.

Weiterhin müssen wir die Anzahl der nun umgebungsvollständigen und umgebungsfragmentarischen Werte aktualisieren. Die Anzahl der zusätzlich geladenen Werte erhöht die Anzahl der umgebungsvollständigen Werte:

$$v_{vc|k|\tau+} = v_{vc|k|\tau} + v'_{l|k|\tau}.$$

Die Anzahl der umgebungsfragmentarischen Werte wird allerdings verringert, und zwar um die Anzahl der zusätzlich induzierten aber nicht zusätzlich geladenen Werte. Diese Werte haben gewissermaßen die vorher umgebungsfragmentarischen Werte „vervollständigt“:

$$v_{vf|k|\tau+} = v_{vf|k|\tau} - v'_{i|k|\tau} + v'_{l|k|\tau}.$$

Die Anzahl der ungeladenen Werte ergibt sich implizit wie immer aus der der Differenz der Kardinalität der Spalte zur Anzahl der geladenen Werte. Ebenfalls ergibt sich diese Anzahl wiederum aus der Summe der Anzahl der umgebungsfragmentarischen und umgebungsvollständigen Werte:

$$v_{u|k|\tau+} = c_k - v_{l|k|\tau+} = c_k - v_{vf|k|\tau+} - v_{vc|k|\tau+}.$$

5.7 Erzeugung neuer Token und Schmuggeln von Werten

Während eines Füllschritts werden natürlich auch Werte in die Schmuggelziele der Cache-Tabelle eingeführt. Dieses Schmuggeln wird durch Erzeugungen modelliert, wenn das Schmuggelziel ein Erzeuger ist.

Anders ausgedrückt: Wir generieren in diesem Fall ein neues Token, das die zusätzlich in die Spalte geladenen Werte repräsentiert. Die Werte wurden während des Füllschritts geschmuggelt und waren zuvor in der Spalte nicht geladen.

Aber auch wenn das Schmuggelziel kein Erzeuger ist, so hat das Schmuggeln von Werten Einfluss auf das Füllverhalten. Durch das Einführen der Werte sind nach dem Füllschritt mehr Werte geladen; diese beeinflussen also die Anzahl der anschließend noch zusätzlich zu ladenden Werte. Werte, die während einem Füllschritt in das Schmuggelziel geschmuggelt wurden, können anschließend nicht mehr zusätzlich geladen werden. Wir müssen also in jedem Fall, unabhängig davon, ob das Schmuggelziel ein Erzeuger ist oder nicht, die Statistik in dieser Spalte aktualisieren.

Wir wollen nun betrachten, wie man die Anzahl $v'_{l|z|\tau}$ der in ein Schmuggelziel z zusätzlich geladenen Werte abschätzen kann, wenn diese Spalte beim Füllschritt nicht aktiv war. Von

der Spalte k wird also ein Token verarbeitet, die Cache-Tabelle T(k) macht also einen Füllschritt. Zwischen k und z existiert eine Schmuggler-Kante $k \rightsquigarrow z$ im SBN.[29]

Um die Größe $v'_{l|z|\tau}$ abzuschätzen, müssen wir wieder eine obere Schranke finden. Die Anzahl der zusätzlich geladenen Werte hängt natürlich von der Anzahl der während des Füllschritts in die Cache-Tabelle geladen Sätze ab.

Jeder Satz liegt in genau einer Werteumgebung. Die Anzahl der Werte, die unter den geladenen Sätzen ist, ist also maximal so groß wie die Anzahl der Sätze selbst:

$$v'_{l|z|\tau} \leq s'_{T(k)|\tau}.$$

Auf der anderen Seite wird die Anzahl der zusätzlich geladenen Werte wie immer durch die Anzahl der noch ungeladenen Werte beschränkt:

$$v'_{l|z|\tau} \leq v'_{u|z\tau}.$$

Wegen der Worst-Case-Abschätzung schätzen wir nun die Anzahl der zusätzlich geladenen Werte durch das Minimum ab:

$$v'_{l|z|\tau} = \min(v'_{u|z|\tau}, s'_{T(k)|\tau}).$$

Nun stellt sich die Frage, wie viele der zusätzlich geladenen Werte umgebungsvollständig sind und wie viele lediglich umgebungsfragmentarisch. Der Einfachheit halber und wegen der Worst-Case-Analyse gehen wir davon aus, dass so viele der zusätzlich geladen Werte wie möglich umgebungsfragmentarisch sind, denn diese Werte können zu einem weiteren Laden von Sätzen führen, wenn sie in einem nachfolgenden Füllschritt „vervollständigt“, also zusätzlich induziert werden. Wir nehmen also an, dass die Gleichung

$$v'_{vf|z|\tau} = v'_{l|z|\tau}$$

gilt, falls in der Spalte überhaupt umgebungsfragmentarische Werte sein können. Beispiel für eine Ausnahme wäre eine Unique-Spalte, denn hier können keine Werte umgebungsfragmentarisch sein. In diesem Fall gilt natürlich

$$v'_{vc|z|\tau} = v'_{l|z|\tau}.$$

Des Weiteren ergibt sich die Frage, wie viele der während des Füllschritts geschmuggelten Werte dazu führen, dass vorher umgebungsfragmentarische Werte nach dem Füllschritt umgebungsvollständig sind. Wie wir gesehen haben, nehmen wir auf Grund der Worst-Case-Analyse an, dass so viele Werte wie möglich zusätzlich geladen werden. Ist aber die Anzahl der ungeladenen Werte kleiner als die Anzahl der während des Füllschritts geladen Werte, so müssen einige Sätze in der Werteumgebung umgebungsfragmentarischer Werte liegen.

Der Einfachheit halber nehmen wir an, dass durch das Laden von Sätzen in die Cache-Tabelle keine Werte im Schmuggelziel „vervollständigt“ werden. Dies ist ebenfalls im Sinne der Worst-Case-Analyse, denn auf diese Art und Weise kann jeder Wert noch zu einem Laden von zusätzlichen Sätzen genutzt werden. Die einzige Bedingung, die stets beachtet werden

[29] Zur Erinnerung: auch im SBN existieren Schmuggler-Kanten, wenn diese im SBG der Cache-Group bzw. -Föderation vorhanden sind.

muss, ist, dass Unique-Spalten überhaupt keine umgebungsfragmentarischen Werte haben können.

Wir haben nun die vollständige Information, um die Statistik im Schmuggelziel zu aktualisieren und ein neues Token zu generieren.

Das neue Token hat natürlich die Historie

$$(\tau, (z, v'_{l|z|\tau})),$$

denn es wurde zum Erzeugungszeitpunkt τ in der Erzeugungsspalte z generiert und repräsentiert zunächst $v'_{l|z|\tau}$ Werte.

Bei Nicht-Unique-Spalten muss die Anzahl der umgebungsfragmentarischen Werte erhöht werden:

$$v_{vf|z|\tau+} = v'_{vf|z|\tau} + v'_{l|z|\tau}.$$

Analog dazu ändert sich bei Unique-Spalten die Anzahl der umgebungsvollständigen Werte:

$$v_{vc|z|\tau+} = v'_{vc|z|\tau} + v'_{l|z|\tau}.$$

5.8 Überblick über einen Füllzyklus

Wir haben nun das notwendige Wissen, um zu sehen, wie ein Füllzyklus im Modell nachgeahmt wird. Dazu betrachten wir zunächst einen einzelnen Füllschritt und sehen dann, wie der Füllzyklus beginnt, Füllschritte ausgeführt werden und der Algorithmus terminiert.

Ein Füllschritt zum Zeitpunkt τ beginnt mit der Auswahl eines Tokens durch den Token-Manager. Der Konsument k am Ende der RCC, in deren Stelle das Token liegt, wird aktiv und verarbeitet das Token. Ohne Beschränkung der Allgemeinheit habe das Token die Historie

$$(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow \ldots \rightarrow (c_{n-1}, v'_{l|c_{n-1}|\tau_{n-1}})).$$

Zunächst wird die Anzahl der zusätzlich geladenen und zusätzlich induzierten Werte berechnet. Hierfür wird erst geprüft, ob das ankommende Token im Konsumenten k erzeugt wurde, also ob $k = c_1$ gilt. In diesem Fall sind alle ankommenden Werte bereits geladen und können höchstens zusätzlich induziert werden. In diesem Fall liegen die ankommenden Werte aber auch alle in der DBD des Konsumenten. Somit gilt in diesem Fall:

$$v'_{l|k|\tau} = 0$$

und

$$v'_{i|k|\tau} = \min(v_{u|k|\tau} + v_{vf|k|\tau},\ v'_{l|c_{n-1}|\tau_{n-1}}).$$

Im anderen Fall, wenn das Token also in einer anderen Spalte als k erzeugt wurde, gilt:

$$v_{i|k|\tau} = \min(v'_{l|c_1|\tau_1} - v_{k1}, \ldots, v'_{l|c_i|\tau_i} - v_{ki},\ v_{u|k|\tau} + v_{vf|k|\tau},\ |V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1} \rightarrow k|\tau},\ v'_{l|c_{n-1}|\tau_{n-1}})$$

und

$$v_{l|k|\tau} = \min(v'_{l|c_1|\tau_1} - v_{k1}, \ldots, v'_{l|c_i|\tau_i} - v_{ki},\ v_{vf|k|\tau},\ |V_{c_{n-1}} \cap V_k| - v_{i|c_{n-1} \rightarrow k|\tau},\ v'_{l|c_{n-1}|\tau_{n-1}}).$$

Dabei wurden die oberen Schranken $v'_{l|c_1|\tau_1}$ -$v_{\hbar 1}$,…, $v'_{l|c_i|\tau_i}$ -$v_{\hbar i}$ mit Hilfe des Kontext-Baums mit der ID des ankommenden Tokens ermittelt.

Auf Grund dieser Daten und der aktuellen Statistik wird die Anzahl der beim betrachteten Füllschritt in die Cache-Tabelle geladenen Sätze berechnet:

$$s'_{T(k)|\tau} = \min(s'_{i|k|\tau} - v'_{i|k|\tau} + v'_{l|k|\tau}, |S_{T(k)}| - s_{l|T(k)|\tau}).$$

Außerdem müssen wir die Statistik im Konsumenten aktualisieren:

$$v_{vc|k|\tau+} = v_{vc|k|\tau} + v'_{l|k|\tau}$$

und

$$v_{vf|k|\tau+} = v_{vf|k|\tau} - v'_{i|k|\tau} + v'_{l|k|\tau}.$$

Des Weiteren müssen wir durch Integration des angekommenen Tokens den Kontext-Baum aktualisieren.

Im Anschluss daran wird die Historie des Tokens um den Besuch im Konsumenten k verlängert, wenn Werte zusätzlich in k geladen wurden. Die Besuchskette erhält also einen neuen Eintrag und sieht anschließend folgendermaßen aus:

$$(\tau_1, (c_1, v'_{l|c_1|\tau_1}) \rightarrow \ldots \rightarrow (c_{n-1}, v'_{l|c_{n-1}|\tau_{n-1}}) \rightarrow (k, v'_{l|k|\tau}).$$

Das Token muss natürlich entsprechend der Anzahl der ausgehenden RCCs dupliziert werden. In jede Stelle einer ausgehenden RCC wird dann ein Duplikat des Tokens eingefügt.

Nachdem wir diese Größe berechnet haben, können wir ermitteln, wie viele Werte in den Schmuggelzielen der Cache-Tabelle zusätzlich geladen werden:

$$v'_{l|z|\tau} = \min(v'_{u|z|\tau}, s'_{T(k)|\tau}).$$

Bei Nicht-Unique-Spalten trägt diese Anzahl zur Erhöhung der umgebungsfragmentarischen Werte bei, und bei Unique-Spalten wird die Anzahl der umgebungsvollständigen Werte erhöht.

Auch von den Schmuggelzielen wird ein Token erzeugt, wenn sie mindestens eine ausgehende RCC haben, also ein Erzeuger sind. Das Token hat dann jeweils die Historie

$$(\tau, (z, v'_{l|z|\tau})).$$

Ablauf eines Füllzyklus

Nachdem wir nun wissen, wie ein einzelner Füllschritt abläuft, wollen wir sehen, wie wir einen Füllzyklus modellieren.

Zu Beginn müssen natürlich die Statistiken in allen Spalten im SBN initialisiert werden. Nach Voraussetzung gehen wir davon aus, dass die Cache-Tabellen vor dem zu betrachtenden Füllzyklus alle leer sind. Es gilt also insbesondere

$$v_{vc|c|\alpha} = 0 \quad \text{und} \quad v_{vf|c|\alpha} = 0 \quad \text{für alle Spalten c im SBN.}$$

Daraus ergibt sich natürlich, dass auch $s_{|T}$ für alle Cache-Tabellen T ist. Auch dieser Wert wird vor dem Füllzyklus initialisiert. Prinzipiell sind hier natürlich auch andere Größen zur Initialisierung möglich, falls dies von Interesse ist.

Nachdem die Statistiken in den Spalten initialisiert sind, wird das initiale Token in die Stelle zwischen einem Cache-Key und seiner Kontrollspalte eingefügt. Das Token hat die Historie

$$(\alpha, (d, 1)).$$

Dabei ist α der Zeitpunkt vor dem ersten Füllschritt und d die Kontrollspalte des Cache-Keys. Wir betrachten einen einzelnen neuen Cache-Key-Wert, der beim ersten Füllschritt beim Cache-Key ankommt.

Nun wird das initiale Token in die Stelle vor dem gewähltem Cache-Key eingefügt. Der Füllzyklus beginnt mit der Verarbeitung dieses initialen Tokens.

Nach dem ersten Füllschritt werden iterativ solange weitere Füllschritte ausgeführt, bis kein Token mehr im SBN zirkuliert. Wir können davon ausgehen, dass selbst bei Cache-Groups und -Föderation, die nicht sicher sind, bei denen es also zu einem rekursiven Laden kommen kann, irgendwann keine neue Token mehr erzeugt werden. Die Erzeuger einer Cache-Tabelle generieren spätestens dann keine weiteren Token mehr, wenn in der Cache-Tabelle alle Sätze aus dem Backend geladen sind. Ebenso werden dann keine Token mehr weiter gegeben, da keine Werte zusätzlich geladen werden. Sogar bei exzessivem Füllverhalten ist dieser Zustand irgendwann erreicht.

Jedes Token hat eine begrenzte Lebenszeit. Spätestens wenn ein Token bei einer Spalte ankommt, die keine ausgehenden RCCs hat, wird es verworfen. Dies kann natürlich schon früher der Fall sein, nämlich wenn das Token eine Spalte besucht, dort aber nicht zu einem zusätzlichen Laden von Sätzen führt. Die Anzahl der zusätzlich geladenen Werte ist dann gleich Null und das Token wird ebenfalls verworfen.

Wir können also davon ausgehen, dass irgendwann keine Token mehr im SBN zirkulieren und der Algorithmus zur Modellierung des Füllverhaltens somit terminiert.

Nachdem die betrachtete Cache-Group bzw. -Föderation diesen Zustand erreicht hat, also stabil ist, können wir die Füllstände der Cache-Tabellen auswerten und für eine quantitative Analyse der Cache-Group in einem Kostenmodell (z. B. dem in Abschnitt 1.7) benutzen.

6 Evaluierung

In den vorangegangenen Abschnitten haben wir Modelle zur Analyse des Füllverhaltens für drei verschiedene Klassen von Cache-Groups bzw. -Föderationen vorgestellt. Wir wollen nun einen Ansatz dafür vorstellen, wie man die „Güte“ der entwickelten Modelle unter bestimmten Voraussetzungen bewerten kann. Wir verstehen unter der Güte eines Modells die Genauigkeit der gelieferten Ergebnisse im Vergleich zum realen Füllverhalten der modellierten Cache-Group bzw. -Föderation. Wir prüfen also immer ein konkretes Modell mit einer konkreten Cache-Group bzw. -Föderation.

Man beachte, dass wir für die gleiche Cache-Group verschiedene Modelle benutzen können: Für eine baumartige Cache-Group steht z. B. ein Modell mit und ein Modell ohne Histogramme zur Verfügung, wir können aber auch das allgemeine Modell (aus Kapitel 0) verwenden. Andererseits können wir das gleiche Modell auch anhand verschiedener Cache-Groups bewerten.

Für das hier vorgeschlagene Evaluierungsverfahren gehen wir von der Gleichverteilungsannahme aus und verzichten auf die Betrachtung von Null-Werten in den Spalten. Außerdem benutzen wir die in Abschnitt 2.8.2 vorgestellte RCC-Teilmengen-Annahme.

Man beachte, dass wir in manchen Modellen von diesen Annahmen absehen. So benutzen wir in den vorgestellten Modellen meistens eine konkrete Überdeckung und verwenden die RCC-Teilmengen-Annahme nicht. Eine Evaluierung ist also für ein solches Modell nur möglich, wenn die RCC-Teilmengen-Annahme in der modellierten Cache-Group gilt. Gleiches gilt für die Gleichverteilungsannahme: Für baumartige Cache-Groups steht ein Modell zur Verfügung, welches Histogramme zur Berechnung der Füllstände verwendet. Dieses Modell kann mit dem hier vorgestellten Verfahren zur Evaluierung nicht bewertet werden, da die Information der Histogramme nicht verwendet werden kann.

Eine optimale Evaluierung würde natürlich mit real vorhandenen Daten arbeiten und so die in dieser Arbeit vorstellten Modelle testen. Dies könnte geschehen, indem man eine real existierende Cache-Verwaltungssoftware (z. B. DBCache) einsetzt und einen Füllzyklus auslöst, indem man einen Cache-Miss provoziert. Anschließend könnten die in die Cache-Tabellen geladen Sätze manuell „abgezählt“ werden. Natürlich würde dieses Verfahren mehrmals wiederholt, wobei wir den Cache-Key-Wert variieren, der den festgelegt Cache-Miss provoziert, um so einen Mittelwert für die Füllstände zu erhalten. Die gewonnenen Daten über das reale Füllverhalten der Cache-Group würden dann mit der modellhaften Analyse verglichen. Auf diese Art und Weise könnte die Genauigkeit der Modelle effizient geprüft werden.

6.1 Population von Backend-Tabellen

Da die Modelle eine gewisse Allgemeingültigkeit besitzen und für die Evaluierung oftmals reale Daten nicht zur Verfügung stehen, soll in diesem Abschnitt ein Verfahren gezeigt werden, wie solche Daten für die Population der Backend-Tabellen zur Evaluierung generiert werden können. Diese Daten haben folgende Eigenschaften:

- Die Tabelle T hat nach der Population eine vorgegebene Kardinalität $|S_T|$,
- in jeder Spalte k der Backend-Tabelle T kommen eine vorgegebene Anzahl c_k unterschiedlicher Werte vor und
- die Häufigkeit $h_k(\varphi)$ der Werte φ ist zwar nicht genau gleich groß für jeden Wert $\varphi \in V_k$ (gemäß der Gleichverteilungsannahme müsste $h_k(\varphi) = |S_T|/c_k$ gelten), die Wahrscheinlichkeit $p(\eta \in S_{k,\varphi})$, mit einem gewählten Satz η aus der Backend-Tabelle T einen bestimmten Wert φ in einer gegebenen Spalte k zu „ziehen“, ist aber näherungsweise gleichverteilt. Durch diese Eigenschaft soll die Gleichverteilungsannahme möglichst genau angenähert werden.

Da das Füllverhalten einer Cache-Group bzw. -Föderation nicht von den Datentypen in den Spalten abhängt, können wir der Einfachheit halber einheitlich in allen Spalten Integer-Werte verwenden. Die Spalten-Kardinalität wird jeweils entsprechend der zu modellierenden Cache-Group gewählt, gleiches gilt für die Kardinalität der Tabelle. Durch das hier vorgestellte Verfahren zur Population der Backend-Tabellen werden diese „Ziel-Kardinalitäten“ immer genau erreicht.

Um das Verfahren so einfach wie möglich zu halten, verwenden wir als Daten zur Population die Werte am Anfang des Zahlenstrahls. Wenn die Kardinalität einer Spalte z. B. 100 ist, so wird die Tabelle mit Sätzen so gefüllt, dass in der Spalte die Werte 1 bis 100 vorkommen. Man beachte, dass hierdurch die Spaltenunabhängigkeitsannahme nicht mehr gewährleistet wird. Auf der anderen Seite gilt durch die gewählte Menge der Daten auf den Fall die RCC-Teilmengen-Annahme.

Mit Hilfe dieses Verfahrens zur Population der Tabellen können ebenfalls leider keine bestimmten Überdeckungen zwischen den Spalten generiert werden. Für eine Gewährleistung dieser Eigenschaften müssten wir die Tabellen mit einem wesentlich aufwendigeren Verfahren füllen.

Das vorgeschlagene Verfahren arbeitet folgendermaßen: In einer Schleife werden gemäß der vorgegebenen Kardinalität der Tabelle Datensätze erzeugt, wobei die Werte in den Spalten per Zufallsgenerator erzeugt werden. Die Werte liegen dabei für die eine Spalte k im Intervall zwischen 1 und c_k. Die Tabelle erhält einen gesonderten Primärschlüssel, mit der wir die Sätze identifizieren können. Wie bereits erwähnt verwenden wir Integer-Werte als Datentypen der Spalten.

Wir nennen diesen Schritt im Füllverfahren, bei der erstmalig Sätze in die Backend-Tabelle eingeführt werden, die **initiale Population**.

Nachdem eine Tabelle wie beschrieben gefüllt ist, wird geprüft, ob die Kardinalitätsrestriktionen für die Spalten gelten. Schließlich ist es sicherlich möglich, dass manche Werte zufällig öfter als mit der durchschnittlichen Häufigkeit $|S_T|/c_k$ (gemäß der Gleichverteilungsannahme)

gewählt wurden und dafür andere Werte überhaupt nicht in der Spalte vorkommen. Dadurch ist im allgemeinen die Kardinalität der Spalte niedriger als vorgesehen.

Die Tabelle wird also nochmals durchlaufen und dabei die vorhandenen Werte geprüft. Nach diesem so genannten **Prüfschritt** für eine bestimmte Spalte wissen wir, welche Werte in der Spalte fehlen. Dabei halten wir gleichzeitig fest, welche Werte mehrfach in der Spalte vorkommen.

Wir müssen nun die fehlenden Werte ergänzen. Dazu gehen wir die fehlenden Werte der Reihe nach durch und führen sie in die Spalte ein. Dafür ersetzen wir ein Vorkommen eines mehrfach vorhandenen Werts durch den einzuführenden Wert. Nehmen wir an, wir wollen den Wert φ einführen, da er vorher noch nicht in der dynamischen Domain (das ist die Menge der vorkommenden Werte) der Spalte war. Damit wir nicht durch das Einführen des Werts einen anderen Wert verdrängen (die Kardinalität $|S_T|$ der Tabelle darf sich schließlich nicht ändern), wählen wir einen Wert φ', der mehrfach in der Tabelle vorkommt. Damit verbunden suchen wir einen Satz, der diesen Wert in der betrachteten Spalte hat. Wir wissen, dass mehrere solche Sätze existieren, denn wir haben mit φ' einen Wert gewählt, der mehrfach vorkommt. Aus der Werteumgebung von φ wählen wir also zufällig einen Satz aus. Wir ersetzen den Wert in der betrachteten Spalte im gewählten Satz mit dem einzuführenden Wert φ. Die Einführung dieses vorher fehlenden Werts φ in die Spalte nennen wir **Korrekturschritt**.

Nachdem wir für jeden zuvor fehlenden Wert einen Korrekturschritt gemacht haben, untersuchen wir die nächste Spalte auf fehlende Werte. Wir machen also einen erneuten Prüfschritt. Auf diesen Prüfschritt folgt wieder eine Reihe von Korrekturschritten.

Wir führen für jede Spalte in der Tabelle einen Prüfschritt aus und machen anschließend jeweils eine Reihe von Korrekturschritten. Nachdem wir jede Spalte korrigiert haben, hat die Tabelle die geforderten Eigenschaften bezüglich Kardinalität der Spalten und Anzahl der vorhandenen Sätze.

6.2 Ablauf der Evaluation, Implementierung des Verfahrens

Nachdem wir Daten entsprechend der modellierten Cache-Group erzeugt haben, können wir einen Füllzyklus simulieren. Dafür generieren wir einen zweiten Satz von Tabellen. Diese zusätzlichen Tabellen, die zu Beginn leer sind, spielen die Rolle der Cache-Tabellen, wohingegen die zuvor gefüllten Tabellen die Backend-Tabellen darstellen.

Wir haben nun die statischen Strukturen zur Evaluierung einer Cache-Group zur Verfügung. Um die Genauigkeit eines Modells zu prüfen, führen wir mehrfach Füllzyklen für unterschiedliche Cache-Key-Werte aus und leeren dazwischen jeweils die Cache-Tabellen immer wieder. Nach jedem Füllzyklus zählen wir die Anzahl der in den Cache-Tabellen geladenen Sätze und merken uns diese Füllstände. Nach einer gewissen Anzahl von Füllzyklen generieren wir die Belegungen der Backend-Tabellen neu, um nicht per Zufall eine Abweichung von den modellierten Füllständen zu erhalten, die nur von der Belegung der Backend-Tabellen herrührt. Durch die sporadische neue Population der Backend-Tabellen erhalten wir ein „durchschnittlicheres" Muster. Hauptsächlich wollen wir dadurch bewirken, dass der Einfluss der nicht

genau erreichten Gleichverteilung möglichst gering wird. Außerdem wird eine mögliche zufällige Abhängigkeit zwischen den Werten in den Spalten einer Tabelle soweit es geht verringert.

Nachdem wir eine Reihe von Füllzyklen mit sporadischen Neubelegungen der Backend-Tabellen ausgeführt haben, vergleichen wir die Mittelwerte der Füllstände mit den vom Modell gelieferten Werten. Wir erhalten auf diese Art und Weise einen Anhaltspunkt für die Güte des Modells.

Die Ausführung eines Füllzyklus wurde für die Evaluierung in sehr einfacher Art und Weise implementiert. Genau wie in Kapitel 0 verwenden wir Token, um zu signalisieren, dass weitere Füllschritte ausgeführt werden müssen. Im Gegensatz zu Kapitel 0 stellen die Token hier aber keine Wertemengen dar, sondern signalisieren lediglich, dass die zur Stelle gehörende RCC möglicherweise nicht gilt und gegebenenfalls vom Nachfolger ein Füllschritt ausgeführt werden muss. Die Token werden wie gehabt in Stellen gespeichert und von einem Token-Manager verwendet, um zu entscheiden, welche Cache-Tabellen einen Füllschritt ausführen müssen.

Betrachten wir einen Füllzyklus Schritt für Schritt. Wir beginnen mit einem leeren Satz von Cache-Tabellen und keinem Token. Zunächst wird die Werteumgebung eines zufällig gewählten Cache-Key-Werts von der Backend-Tabelle in die Cache-Tabelle geladen. Jede Spalte in dieser Cache-Tabelle prüft nun, ob neue Werte eingeführt wurden. Im Gegensatz zur Modellierung sind hier aber keine abschätzenden Berechnungen notwendig, da alle „fließenden“ Werte und Wertemengen bekannt sind. Wenn neue Werte eingeführt wurden, so wird in die Stellen der ausgehenden RCCs ein Token eingefügt.[30]

Dieses Token signalisiert der Nachfolger-Tabelle, dass von ihr ein Füllschritt ausgeführt werden muss. Insofern reicht es hier, wenn wir pro Stelle ein Token verwalten. Wenn wir in einer Stelle bereits ein Token finden, so werden keine weiteren Token mehr eingefügt.

Nach jedem Füllschritt wird vom Token-Manager geprüft, ob noch weitere Token vorhanden sind. Falls ja, so wird ein beliebiges Token gewählt und entnommen. Die Cache-Tabelle am Ende der betreffenden RCC führt daraufhin einen Füllschritt aus.

Bei einem allgemeinen Füllschritt wird die Werteumgebung der Menge der im Vorgänger geladen Werte in die Cache-Tabelle geladen. Man beachte, dass bei diesem Verhalten eventuell manche Sätze mehrfach von der Backend-Tabelle in die Frontend-Tabelle kopiert werden, da nicht zuvor geprüft wird, ob der Satz bereits in der Cache-Tabelle ist. Dennoch wird verhindert, dass ein Satz mehrfach in den Cache geladen wird. Diese Prüfung geschieht mit Hilfe der Eindeutigkeit der Sätze, die durch den (zusätzlich hinzugefügten) Primärschlüssel gegeben ist.

Da wir das System nur zur Evaluierung eines Modells und nicht für den laufenden Betrieb benötigen, können wir auf Performance-Optimierungen verzichten. Für eine Implementierung

[30] Genau genommen wird dies von der Cache-Verwaltung geprüft, aber der Einfachheit halber sagen wir, dass die Spalte die Prüfung ausführt. Dies entspricht auch einem objekt-orientierten Ansatz.

eines Caches wären an dieser Stelle aber mit Sicherheit noch Verbesserungen möglich und notwendig.

Bei einem Füllschritt wird wieder geprüft, ob neue Werte in die Spalten eingeführt wurden. Ist dies der Fall, so werden zusätzliche Token generiert und in die nachfolgenden Stellen eingefügt.

Der Füllzyklus ist dann beendet, wenn keine Token mehr gefunden werden. Wir können dann davon ausgehen, dass alle RCCs erfüllt sind und die Cache-Group somit stabil ist.

6.3 Beispielhafte Evaluierung

Wir haben das oben dargestellte Verfahren dazu verwendet, um den Algorithmus ohne Histogramme für baumartige Cache-Groups beispielhaft zu evaluieren. Die untersuchte Cache-Group wurde in Abschnitt 3.4.2 vorgestellt und in Abbildung 13 gezeigt. Es handelt sich dabei um eine Cache-Group, bei der die RCC-Teilmengen-Eigenschaft erfüllt ist.

Wir haben mit dem Algorithmus aus Abschnitt 3.4.1 die durchschnittlichen Füllstände nach einem einzelnen Füllzyklus abgeschätzt. Der Algorithmus lieferte die Ergebnisse, wie sie in Tabelle 2 dargestellt sind.

Cache-Tabelle (Name)	Füllstand (Anzahl geladener Sätze)
A	5
B	4,976
C	5

Tabelle 2: Ergebnisse des Modells für baumartige Cache-Groups ohne Histogramme

Wir haben das Modell für die gleiche Cache-Group mit dem oben dargestellten Verfahren evaluiert. Dafür haben wir zunächst eine Belegung für die drei Backend-Tabellen erzeugt, wie sie vorgegeben waren. Anschließend haben wir 20 Füllzyklen ausgeführt und jeweils die Füllstände „gemessen“. Nach dem zehnten Füllzyklus haben wir eine neue Belegung für die Backend-Tabellen erzeugt.

Die Ergebnisse zeigt Tabelle 3. Wie zu erwarten war, weichen die realen Ergebnisse etwas von den Ausgaben des Modells ab. Was dabei auffällt, ist, dass die realen Füllstände der drei Tabellen vom Modell systematisch unterschätzt werden. Eine Vermutung dafür ist, dass im Modell mit Fließkomma-Werten (Erwartungswerten und Durchschnittswerten) gearbeitet wird und dadurch geringere Werte abgeschätzt werden. Allerdings halten sich die Abweichungen durchaus in vertretbaren Grenzen, so dass wir die modellhafte Analyse durchaus als brauchbar einschätzen können.

Das vorgestellte Beispiel sagt allerdings noch recht wenig über die Güte der Modelle im Allgemeinen aus. Lediglich das baumartige Modell ohne Histogramme wurde hier bewertet. Für eine Evaluation der anderen Modelle müsste aber zunächst das jeweilige Modell implementiert werden. Eine solche Implementierung der verschiedenen Modelle steht aber noch aus.

Wir haben im Rahmen dieser Arbeit das Verfahren zur Evaluation als eine Bibliothek von Java-Klassen implementiert. Diese können unter [Bay04b] herunter geladen werden und als Grundlage für weitere Bewertungen dienen.

Nummer des Füllzyklus	Füllstand A	Füllstand B	Füllstand C
1	4	4	4
2	4	3	4
3	9	8	9
4	5	5	5
5	5	5	5
6	4	3	4
7	7	7	7
8	8	8	8
9	5	5	5
10	5	5	5
11	3	3	3
12	7	7	7
13	4	4	4
14	3	3	3
15	6	6	6
16	10	10	10
17	3	3	3
18	2	2	2
19	7	6	7
20	4	4	4
Mittelwert	5,25	5,05	5,25

Tabelle 3: Ergebnisse der Evaluierung

7 Zusammenfassung und Ausblick

In dieser Arbeit haben wir das Füllverhalten verschiedener Arten von Cache-Groups und -Föderationen untersucht, und dabei Modelle zur quantitativen Analyse entwickelt. Nach einer allgemeinen Einführung in die Thematik des Datenbank-Caching – und insbesondere des Constraint-basierten Caching und Cache-Groups für Gleichheitsprädikate – in Kapitel 1 wurden in einem Grundlagenteil zunächst generelle Eigenschaften von Cache-Groups untersucht, die für die Modellierung des Füllverhaltens notwendig bzw. hilfreich sind.

7.1 Grundlagenteil

Die Grundlagen für die Entwicklung der Modelle haben wir in Kapitel 2 betrachtet und dabei unter anderem Füllschritte und Zeitpunkte, RCCs, WMPe und Schmuggler-Beziehungen angeschaut.

Zur Modellierung des Füllverhaltens einer Cache-Group oder -Föderation verwenden wir immer wieder Annahmen zur Vereinfachung der Berechnungen. Diese Modellannahmen haben wir definiert und kurz betrachtet.

Außerdem haben wir uns mit dem Spalten-Beziehungsgraph (SBG) einer Cache-Group bzw. -Föderation beschäftigt. Der SBG repräsentiert die wichtigsten Elemente der Cache-Groups, d. h. die Spalten. Außerdem werden die Zusammenhänge zwischen den Spalten entsprechend im SBG dargestellt. Mit Hilfe des SBG haben wir mögliche Zerlegungen der Menge der Spalten einer Cache-Group beleuchtet und dabei die wichtigen Begriffe „Konsument“, „Erzeuger“, „Basiserzeuger“, „Schmuggelquelle“ und „Schmuggelziel“ gebildet. Diese Begriffe helfen uns bei der Formulierung verschiedener Zusammenhänge, die das Füllverhalten einer Cache-Group oder -Föderation betreffen.

Grundlegende Ereignisse beim Füllverhalten einer Cache-Group oder -Föderation sind Füllschritte. Diese können wir in den verschiedenen Modellen aus unterschiedlichen Granularitätsebenen betrachten. Wählen wir eine „feinere“ Granularität, so können wir allgemein in einem Modell eine höhere Genauigkeit der Ergebnisse (d. h. der berechneten Füllstände) erreichen. Auf der anderen Seite ist die Effizienz (im Sinne von Laufzeit) eines Algorithmus aber höher, wenn wir eine „gröbere“ Granularität betrachten. Zwischen Effizienz und Genauigkeit gibt es also einen Zielkonflikt, der von der betrachteten Granularitätsebene beeinflusst wird. Da alle Algorithmen zur Berechnung der Füllstände in einer Cache-Group oder -Föderation Füllschritte betrachten, ist auch hier die Anzahl und Reihenfolge der Füllschritte wichtig, was wir ebenfalls untersucht haben.

Da bei der Modellierung des Füllverhaltens von Cache-Groups und -Föderationen oft die Mächtigkeit einer Schnittmenge berechnet werden muss, haben wir in einem Exkurs Betrachtungen zur Abschätzung der Schnittmengenmächtigkeit gemacht.

Wir konnten zwei maßgebliche Einflussfaktoren auf das Füllverhalten einer Cache-Group bzw. -Föderation identifizieren:

- Über RCCs kommen Werte bei einem Konsumenten an und beeinflussen das Füllverhalten somit über Wirkzusammenhang I.
- Über Schmuggler-Beziehungen (im SBG dargestellt durch Schmuggler-Kanten) werden bei einem Füllschritt Werte in die Schmuggelziele der Cache-Tabelle eingeführt. Diese beeinflussen das Füllverhalten der eigenen Cache-Tabelle, weil in nachfolgenden Füllschritten weniger Werte in das Schmuggelziel (in seiner Rolle als Konsument) zusätzlich geladen werden. Außerdem beeinflussen sie das Füllverhalten nachfolgender Cache-Tabellen, weil abgehende RCCs durch das Schmuggeln der Werte eventuell ungültig werden, so dass die Nachfolger der Schmuggelziele gegebenenfalls ungesättigt sind.

Im Zusammenhang mit RCCs haben wir verschiedene Aspekte untersucht, die Einfluss auf das Füllverhalten haben. Darunter war die RCC-Eigenschaft, aus der wir die RCC-Implikation abgeleitet haben. Die RCC-Implikation besagt, dass die Tatsache, dass ein Wert in einem Erzeuger geladen und in der DBD seines Nachfolgers vorhanden ist, impliziert, dass der Wert nach dem Füllzyklus im Nachfolger induziert ist. Des Weiteren haben wir gezeigt, dass die Überdeckung zwischen zwei Spalten ebenfalls einen großen Einfluss auf die Anzahl der zu induzierenden Werte hat. Je größer die Überdeckung ist, desto mehr der ankommenden Werte liegen in der DBD des Konsumenten und werden damit induziert. Für den Fall, dass wir keine explizite Information über die Überdeckung haben, müssen wir eine Schranke für die Schnittmengenmächtigkeit der DBDs angeben. Diese ergibt sich aus der RCC-Teilmengen-Annahme, die besagt, dass wir davon ausgehen, dass eine „kleinere“ DBD immer eine Teilmenge der „größeren“ DBD ist, wenn ihre Spalten über eine RCC verbunden sind. Wir können dies auch so formulieren, dass wir annehmen, dass zwischen zwei Spalten, die über eine RCC verbunden sind, im Backend eine Fremdschlüssel-Primärschlüssel-Beziehung gilt.

Zusammenhängende Wege von RCC-Kanten nennen wir Wertemengen-Pfade (WMPe). Wertemengen-Pfade bilden einen Pfad, auf dem die Wertemengen in einer Cache-Group bzw. -Föderation „wandern“ und erweitern gewissermaßen die RCC-Implikation für einzelne Werte transitiv: Ein Wert, der in den DBDs aller Spalten entlang des WMPs vorhanden und in der Spalte am Anfang des WMPs geladen ist, wird in die Spalten auf dem WMP induziert. Mit Hilfe von WMPen kann außerdem die Menge der Basiserzeuger eines Konsumenten gefunden werden. Ist der WMP isoliert, so ist der Basiserzeuger eindeutig bestimmt. In diesem Fall ist die Ladewahrscheinlichkeit eines Werts in einem Konsumenten auf dem WMP abhängig von der kombinierten Zugehörigkeitsfunktion und der Ladewahrscheinlichkeit im Basiserzeuger, denn der Wert kann nur dort „erzeugt“ worden sein.

Neben RCCs haben Schmuggler-Beziehungen ebenfalls einen großen Einfluss auf das Füllverhalten von Cache-Groups. Aus diesem Grund widmeten wir unsere Aufmerksamkeit auch dieser Art von Beziehung. Wir haben eine Berechnungsvorschrift für die Wahrscheinlichkeit gefunden, mit der ein Wert in ein Schmuggelziel eingeführt wird und mehrere Möglichkeiten

aufgezeigt, wie man diese Schmuggelwahrscheinlichkeit effizient mit dem Computer berechnen kann. Dabei haben wir die Funktion zur Berechnung der Schmuggelwahrscheinlichkeit so erweitert, dass wir sie auch für reelle Füllstände anwenden können.

7.2 Hauptteil

Im Hauptteil der Arbeit haben wir Modelle für drei verschiedene Klassen von Cache-Groups entwickelt:

- Den einfachsten Fall stellen Cache-Groups dar, deren SBG ein Baum ist. Wir haben dass Füllverhalten dieser so genannten baumartigen Cache-Groups in Kapitel 3 betrachtet.
- Eine zweite Klasse von Cache-Groups stellen die nicht-baumartigen, RCC-zyklenfreien Cache-Groups dar. Der Erreichbarkeitsgraph einer solchen Cache-Group ist ein gerichteter, azyklischer Graph. Wir haben das Füllverhalten dieser Cache-Groups in Kapitel 4 mit Hilfe eines einfachen Modells untersucht.
- Im allgemeinen Fall haben wir es mit der dritten Klasse von Cache-Groups und -Föderationen zu tun. Die betrachteten Cache-Groups bzw. -Föderationen sind hier also nicht mehr RCC-zyklenfrei und deshalb insbesondere auch nicht baumartig. Wir haben ein mögliches Verfahren zur Worst-Case-Analyse des Füllverhaltens einer solchen Cache-Group bzw. -Föderation angeboten.

Die Modelle liefern dabei jeweils Abschätzungen der Füllstände in den Cache-Tabellen, nachdem ein einzelner Füllzyklus eines zuvor leeren Caches gemacht wurde. Da es durch die konkrete Wahl der referenzierten Cache-Keys und Cache-Key-Werte zu unterschiedlichen Füllständen kommen kann, liefern die Modelle durchschnittliche Werte.

7.2.1 Baumartige Cache-Groups

Der SBG einer baumartigen Cache-Group ist ein Baum, d. h. jede Spalte hat maximal einen Vater. Genau genommen hat jede Spalte bis auf die Wurzel genau einen Vater und die Wurzel hat keinen Vater. Außerdem erlauben wir in einer solchen Cache-Group lediglich einen Cache-Key, der die Wurzel des SBGen ist.

Dadurch, dass die SBGen der betrachteten Cache-Groups Bäume sind, ergeben sich angenehme Eigenschaften, die die Modellierung des Füllverhaltens erheblich vereinfachen. So sind in einer baumartigen Cache-Group alle RCCs isoliert und damit auch alle WMPe. Ein Wert kann also lediglich entweder durch Schmuggeln oder Induzieren in eine Spalte eingeführt werden.

Jede Cache-Tabelle in einer baumartigen Cache-Group führt genau einen Füllschritt aus, wenn wir nicht Einzelwert-Füllschritte betrachten. In diesem einzelnen Füllschritt werden Werte in den eindeutig bestimmten Konsumenten induziert und in die Schmuggelziele geschmuggelt.

Das Füllverhalten von baumartigen Cache-Groups wurde in einem ersten einfachen Modell zur Berechnung der Füllstände in der Cache-Group benutzt. Wir sind dabei von der Gleichverteilungsannahme ausgegangen, haben im entwickelten Algorithmus also keine Histogramme benutzt. Der Algorithmus durchläuft die Cache-Tabellen in topologischer Sortierordnung und berechnet dabei die jeweilige Anzahl der in den Konsumenten indizierten Werte anhand der Anzahl der im Vorgänger geladenen Werte. Damit verbunden wird der Füllstand der Cache-Tabelle bestimmt.

Des Weiteren ist dadurch, dass alle WMPe isoliert sind, gewährleistet, dass zu jedem Konsumenten ein eindeutig bestimmter Basiserzeuger gefunden werden kann. Die induzierten Werte stammen also alle aus dem Basiserzeuger des Konsumenten. Mit den in Kapitel 0 eingeführten Begriffen können wir sagen, dass die Werte aus einer Erzeugung im Basiserzeuger stammen und über den WMP bis zum Konsumenten geflossen sind.

Da jeder induzierte Wert aus einem eindeutig bestimmten Basiserzeuger stammt, können wir die Wahrscheinlichkeit berechnen, mit der dieser Wert im Basiserzeuger geladen wurde. Diese Ladewahrscheinlichkeit ist dann zusammen mit der kombinierten Zugehörigkeitsfunktion eine Einflussgröße für die Ladewahrscheinlichkeit im Konsumenten. Mit deren Hilfe und der Häufigkeit des Werts im Konsument lässt sich wiederum ein Erwartungswert für die Anzahl der für ihn geladenen Sätze berechnen. Indem wir einen solchen Erwartungswert für jeden Wert aus der DBD des Konsumenten berechnen und die Erwartungswerte addieren, können wir somit einen Erwartungswert für den Füllstand der Cache-Tabelle berechnen.

Wir haben dieses Verhalten in die Entwicklung eines zweiten Algorithmus zur Berechnung der Füllstände in baumartigen Cache-Groups einfließen lassen. Dieser Algorithmus durchläuft ebenfalls die Cache-Tabellen der Cache-Group in topologischer Sortierordnung, betrachtet aber hier jeden einzelnen Wert aus der DBD des Konsumenten. Anders ausgedrückt verwenden wir Einzelwert-Füllschritte bei der Modellierung des Füllverhaltens. Mit Hilfe der Ladewahrscheinlichkeit wird nun jeweils berechnet, um wie viele Sätze der Füllstand erhöht wird. Nachdem alle Werte betrachtet wurden, steht der Füllstand der betrachteten Cache-Tabelle fest und wir können uns der nächsten Cache-Tabelle zuwenden.

Dieser vorläufig vorgeschlagene Algorithmus ist allerdings sehr schlecht bezüglich seines Laufzeitverhaltens. Das kommt daher, dass in jeder Cache-Tabelle jeder Wert aus der DBD des Konsumenten „angefasst“ werden muss. Das setzt natürlich voraus, dass wir die konkrete DBD der Spalte kennen. Wir können diese Information allerdings über die Histogramme ermitteln. Für die nächste Cache-Tabelle müssen dann wieder alle Werte betrachtet werden usw. Durch diese Modellierung werden manche Werte vermutlich mehrfach betrachtet, für jede Cache-Tabelle erneut.

Wir haben dementsprechend einige Optimierungen zur Steigerung der Effizienz betrachtet. Der letztlich vorgeschlagene Algorithmus dreht die Verschachtelung der Schleifen um. Es wird nicht mehr in jeder Cache-Tabelle jeder Wert in der DBD des Konsumenten betrachtet, sondern die DBD jedes Basiserzeugers wird durchlaufen. Für jeden Wert in der DBD des Basiserzeugers wird nun betrachtet, wie er die Füllstände in den Cache-Tabellen, deren Konsumenten über WMPe mit dem Basiserzeuger verbunden sind, beeinflusst. Durch diese Optimierung müssen die Werte aus der DBD des Basiserzeugers nur ein einziges Mal „angefasst“ werden.

7.2.2 Nicht-baumartige, RCC-zyklenfreie Cache-Groups

Die zweite betrachtete Klasse der Cache-Groups enthält solche, die zwar keine RCC-Zyklen enthalten, aber auch nicht mehr baumartig sind. Wir nehmen aber weiterhin an, dass in der Cache-Group lediglich ein einziger Cache-Key definiert ist und sich dieser in der Wurzel-Tabelle befindet.

Durch die Struktur der Cache-Group können nun an einer Cache-Tabelle prinzipiell mehrere RCC enden, und dies an unterschiedlichen Konsumenten. Unter Betrachtung von RCC-Füllschritten können von einer Cache-Tabelle nun also auch mehrere Füllschritte ausgeführt werden.

Der vorgeschlagene Algorithmus zur Berechnung der Füllstände arbeitet nun ebenfalls mit einer topologischen Sortierordnung der betrachtenden Cache-Tabellen. Jede dieser Cache-Tabellen führt nun für jede eingehende RCC einen Füllschritt aus, wobei wir diese als unabhängige Ereignisse betrachten. Wir gehen also prinzipiell von der Annahme aus, dass die Satzmengen, die während der Füllschritte geladen werden, unabhängig voneinander sind. Des Weiteren verwenden wir im vorgeschlagenen Modell die Gleichverteilungsannahme. Durch die Annahme, dass die Füllschritte unabhängige Ereignisse darstellen, können wir mit Hilfe der Abschätzung der Schnittmengenmächtigkeit (als Average-Case-Analyse) berechnen, wie viele Sätze während eines jeweiligen Füllschritts zusätzlich hinzukommen und so den Füllstand erhöhen.

Nachdem wir den Füllstand der Wurzel-Tabelle in gleicher Weise wie bei baumartigen Cache-Groups abgeschätzt haben, betrachten wir die anderen Cache-Tabellen in topologischer Sortierordnung. Jede Cache-Tabelle führt dabei hintereinander alle RCC-Füllschritte aus, es wird also der Reihe nach jede eingehende RCC betrachtet. Der Algorithmus führt für jede RCC im Prinzip zwei Rechenschritte aus:

- Zunächst wird abgeschätzt, wie viele Werte in den Konsumenten induziert werden. Dies geschieht unter Betrachtung der Anzahl der in den Vorgänger geladenen Werte und der Überdeckung zwischen Vorgänger und Konsument.
- Anschließend wird die Anzahl der während des Füllschritts in die Cache-Tabelle geladenen Sätze berechnet. Dabei fließt in die Berechnung ein, wie viele Sätze vor dem Füllschritt bereits in der Cache-Tabelle waren. Die Schnittmenge zwischen bereits geladenen Sätzen und der Werteumgebung der gerade induzierten Werte trägt natürlich nicht zur Erhöhung des Füllstands bei. Wir verwenden das Prinzip von Inklusion und Exklusion, um die Mächtigkeit der Vereinigung, d. h. den neuen Füllstand, zu berechnen.

Nachdem alle RCCs betrachtet wurden und der Füllstand feststeht, wird die Anzahl der in die Erzeuger eingeführten Werte berechnet. Diese Berechnung geschieht lediglich auf Grund des Füllstands der Cache-Tabelle. Für Konsumenten der Cache-Tabelle wird also nicht zwischen induzierten und geschmuggelten Werten unterschieden. Wir abstrahieren von der Unterscheidung zwischen Wirkzusammenhang I und Wirkzusammenhang II und nehmen implizit an, dass alle Werte in die Erzeuger geschmuggelt wurden. Unter anderem durch diese Annahme wird das Modell ungenau. Weitere Faktoren, die zur Steigerung der Ungenauigkeit beitragen, sind:

- Es wird nicht auf Abhängigkeiten zwischen den Werten geachtet. Wenn ein Konsument über verschiedene WMPe mit dem gleichen Basiserzeuger verbunden ist, so kann ein Wert mehrfach beim Konsumenten ankommen. Natürlich führt das Induzieren dieses Werts nur ein einziges Mal zum Laden von Sätzen aus dem Backend und dadurch zur Erhöhung des Füllstands. Von diesem Zusammenhang wird ebenfalls abstrahiert, da wir von der Unabhängigkeit der Füllschritte ausgehen.
- Natürlich wird auf Grund der Gleichverteilungsannahme ebenfalls eine gewisse Ungenauigkeit verursacht, da wir auf die Betrachtung der Häufigkeit eines Werts in einer Spalte verzichten. Ein Algorithmus, der RCC-Füllschritte mit der Gleichverteilungsannahme betrachtet, ist im Allgemeinen immer ungenauer als ein Algorithmus mit Einzelwert-Füllschritten und Histogrammen, da bei Letzterem mehr Informationen zur Verfügung stehen.

7.2.3 Allgemeine Cache-Groups und -Föderationen

Die dritte betrachtete Klasse von Cache-Groups und -Föderationen umfasst alle, die nicht in den anderen Klassen sind. Wir stellen also an diese Cache-Groups und -Föderationen keine Restriktionen bezüglich ihrer Struktur. Sie können RCC-Zyklen enthalten. Auch sind hier mehrere Cache-Keys in unterschiedlichen Cache-Tabellen erlaubt.

Modellannahmen und Strukturen

Da das Füllverhalten solcher Cache-Groups sehr komplex sein kann, haben wir nur ein Verfahren zur Abschätzung der Füllstände im Worst-Case vorgestellt. Der Einfachheit halber gehen wir in unserem Modell von den in Abschnitt 2.1 vorstellten Modellannahmen aus. Des Weiteren betrachten wir keine Einzelwert-, sondern direkte RCC-Füllschritte.

Das vorgeschlagene Modell arbeitet mit Stellen und Token. Der SBG wird so erweitert, dass jede RCC durch eine Stelle repräsentiert wird. Außerdem modellieren wir die Füllpunkteigenschaft des Cache-Keys durch eine weitere Stelle vor dem Cache-Key. Die Einstiegspunkteigenschaft wird hingegen mit Hilfe einer Selbst-RCC (bzw. einer entsprechenden Stelle) modelliert. Das entstehende Netz aus Spalten, Stellen und Schmuggler-Kanten nennen wir Spalten-Beziehungsnetz (SBN).

Die Token repräsentieren die „fließenden“ Wertemengen. Ein Token wird in einem Erzeuger generiert, wenn neue Werte in ihn eingeführt werden. In Spalten mit mehreren ausgehenden RCCs wird ein verarbeitetes Token dupliziert und in den nachfolgenden Stellen eingefügt. Auf seinem Weg über einen WMP speichert ein Token Daten über die gemachten Besuche in den verschiedenen Spalten.

Durch diese Historie können Abhängigkeiten zwischen ankommenden Wertemengen identifiziert werden. Ebenfalls wird dadurch erreicht, dass wir den Unterschied zwischen homogenen und heterogenen Zyklen unterscheiden können. In einem heterogenen Zyklus werden bei jedem Spalten-Übergang neue Token im Erzeuger bzw. Schmuggelziel generiert und so werden immer mehr Werte geladen. Kommt ein Token allerdings wieder bei seiner Erzeugungsspalte an, so führt dies eventuell zwar zu einer „Vervollständigung“ umgebungsfragmentarischer Werte, aber das Token wird in jedem Fall nach der Verarbeitung verworfen.

Mit Hilfe der Historien von Token mit der gleichen ID (Erzeugungszeitpunkt und Erzeugungsspalte), die zusammengefasst in Kontext-Bäumen gespeichert werden, können wir außerdem feststellen, wie viele der ankommenden Werte bereits induziert wurden. Dies geschieht durch einen Test dahingehend, ob die ankommenden Werte noch in die Kontexte „passen“, ohne dass es zu Überschneidungen mit bereits aus diesem Kontext induzierten Werten (den Kontext-Füllwerten) kommt. Dadurch bekommen wir eine Reihe von oberen Schranken für die Anzahl der zusätzlich induzierten Werte (von denen wegen der Worst-Case-Betrachtung letztendlich die kleinste gilt).

Ablauf eines Füllzyklus

Ein modellierter Füllzyklus läuft nun folgendermaßen ab: Zu Beginn werden die Statistiken in allen Spalten im SBN initialisiert. Nun wird das initiale Token in die Stelle vor dem gewählten Cache-Key eingefügt. Der Füllzyklus beginnt mit der Verarbeitung dieses initialen Tokens. Ein Token-Manager prüft danach und nach jedem Füllschritt, ob noch Token im Netz zirkulieren. Wenn Token gefunden werden, so wird zufällig ein beliebiges Token gewählt und der nachfolgenden Spalte zur Verarbeitung übergeben. Diese Verarbeitung im Konsumenten entspricht einem Füllschritt. Bei diesem Füllschritt werden weitere Token in den Erzeugern generiert, wenn dort neue Werte eingeführt werden. Es werden iterativ solange Füllschritte ausgeführt, bis kein Token mehr im SBN zirkuliert. Nachdem die betrachtete Cache-Group bzw. -Föderation stabil ist, werden die Füllstände der Cache-Tabellen ausgewertet.

Ein Füllzyklus besteht also aus einer Reihe von Füllschritten. Dabei haben wir die Füllschritte folgendermaßen modelliert:

Ein Füllschritt beginnt mit der Auswahl eines Tokens durch den Token-Manager. Der Konsument am Ende der RCC, in deren Stelle das Token liegt, wird aktiv und verarbeitet das Token.

Zunächst wird die Anzahl der zusätzlich geladenen und zusätzlich induzierten Werte berechnet. Hierfür wird zunächst geprüft, ob das ankommende Token im Konsumenten erzeugt wurde. In diesem Fall sind alle ankommenden Werte bereits geladen und können höchstens zusätzlich induziert werden.

Wenn das Token aber in einer anderen Spalte erzeugt wurde, werden die oberen Schranken für die Anzahl der zusätzlich induzierten bzw. zusätzlich geladenen Werte mit Hilfe des Kontext-Baums mit der ID des ankommenden Tokens und der Statistik im Konsumenten ermittelt. In die Berechnung wird die Anzahl der umgebungsvollständigen und umgebungsfragmentarischen Werte einbezogen.

Außerdem wird abgeschätzt, wie viele Werte außerhalb der DBD des Konsumenten liegen. Hierbei spielt die Überdeckung mit der DBD des Vorgängers und die Anzahl der bereits über die RCC induzierten Werte eine Rolle.

Anschließend wird die Anzahl der beim Füllschritt geladenen Sätze abgeschätzt. Hierbei wird ebenfalls zwischen zusätzlich geladenen und lediglich zusätzlich induzierten Werten unterschieden. Letztere führen zu einer geringeren Erhöhung des Füllstands als erstere, da die zusätzlich geladenen Werte dazu beitragen, dass ihre komplette Werteumgebung in den Cache geholt wird. Auf der anderen Seite sind für die umgebungsfragmentarischen Werte bereits

einige Sätze aus ihrer Werteumgebung im Cache, müssen also nicht in die Cache-Tabelle geladen werden. Somit führen auch weniger Sätze zu einer Erhöhung des Füllstands.

Des Weiteren müssen wir durch Integration des angekommenen Tokens den Kontext-Baum aktualisieren. Im Anschluss daran wird die Historie des Tokens um den Besuch im Konsumenten verlängert, wenn Werte zusätzlich geladen werden. Das Token muss natürlich entsprechend der Anzahl der ausgehenden RCCs dupliziert werden. In jede Stelle einer ausgehenden RCC wird dann ein Duplikat des Tokens eingefügt.

Mit dem Wissen über die Anzahl der während des Füllschritts geladenen Sätze können wir ermitteln, wie viele Werte in die Schmuggelziele der Cache-Tabelle zusätzlich geladen werden. Bei Nicht-Unique-Spalten trägt diese Anzahl zur Erhöhung der umgebungsfragmentarischen Werte bei, und bei Unique-Spalten wird die Anzahl der umgebungsvollständigen Werte erhöht. Auch von den anderen Erzeugern wird ein Token erzeugt, wenn Werte in sie geschmuggelt wurden.

7.3 Evaluierung

Nach dem Hauptteil der Arbeit haben wir uns noch kurz mit möglichen Verfahren zur Evaluierung der vorgestellten Modelle beschäftigt. Wir haben dabei unter anderem betrachtet, wie man Belegungen für die Backend-Tabellen erzeugen kann.

Die Population einer Tabelle geschah dabei unter der Voraussetzung, dass die Spalten und die Tabelle vorgegebenen Kardinalitätsrestriktionen genügen sollen. Wir haben also einen Algorithmus dafür entwickelt, die Backend-Tabellen mit Test-Daten zu füllen.

In einem weiteren Schritt haben wir einen Füllzyklus ausgeführt und dabei das Verhalten der modellierten Cache-Group nachgestellt. Anschließend haben wir die Anzahl der in die Cache-Tabellen geladenen Sätze ausgezählt und mit den Modellergebnissen verglichen.

Wir haben beispielhaft das Modell ohne Histogramme für baumartige Cache-Groups evaluiert. Dabei haben wir für die Evaluierung die bereits vorher als Beispiel herangezogene Cache-Group aus Abschnitt 3.4.2 verwendet. Die Ergebnisse der Evaluierung haben einen Hinweis darauf gegeben, dass dieses Modell durchaus für diese einfache Klasse von Cache-Groups verwendbar ist. Zwar gab es kleinere Abweichungen, aber die vom Modell gelieferten Daten konnten gut einen Anhaltspunkt für die Größenordnungen der Füllstände bieten.

7.4 Ausblick

Die in dieser Arbeit entwickelten Modelle berechnen die Anzahl der Sätze in den unterschiedlichen Cache-Tabellen, die durch dass Laden der Werteumgebung eines Cache-Key-Werts und der nachfolgenden Füllschritte in den Cache geladen werden müssen.

Die von den Modellen gelieferten Ergebnisse über die durchschnittlichen Füllstände der Cache-Tabellen können für eine quantitative Analyse des Füllverhaltens der Cache-Group mit Hilfe eines Kostenmodells eingesetzt werden. Das Kostenmodell liefert ein Maß dafür, wie hoch die Replikations- und Wartungskosten der Cache-Group sind. Eine solche quantitative

Analyse ermittelt die Kosten, die durch die Verwaltung und Instandhaltung einer Cache-Group entstehen. In das Kostenmodell fließen neben den zu erwartenden Füllständen auch die Häufigkeiten der Aktualisierung der Tabellen im Backend ein.

Die quantitative Analyse und damit das Kostenmodell sind ein wichtiger Schritt zur Entwicklung eines so genannten Cache-Advisors. Dieses Werkzeug kann einem Datenbank-Administrator Hilfestellung bei der Definition und Wartung von Cache-Groups geben.

Was noch fehlt ist eine konkrete Implementierung eines solchen Hilfsmittels, das einem Datenbank-Administrator zur Definition bestimmter RCCs raten kann. Ein solches Werkzeug könnte die Ergebnisse dieser Arbeit benutzen und auf diesen aufsetzen.

Auf Grund der Komplexität von allgemeinen Cache-Groups und -Föderationen konnte in dieser Arbeit nur ein Modell zur Abschätzung der Füllstände im schlechtesten Fall entwickelt werden. Wir haben lediglich Hinweise darauf gegeben, wie und an welchen Stellen eine Average-Case-Analyse möglich ist. Ein komplettes Modell für eine Analyse des durchschnittlichen Füllverhaltens einer Cache-Group fehlt aber zur Zeit noch.

Des Weiteren sind Optimierungen der Modelle bezüglich ihrer Laufzeit denkbar. Eventuell kann die Anzahl der betrachteten Füllschritte in den Modellen verkleinert werden. Auch die Verringerung der durchschnittlich zirkulierenden Token im allgemeinen Modell kann zu einer Verbesserung des Laufzeitverhaltens führen.

Neben der Laufzeit eines Algorithmus spielt natürlich die Ergebnisgenauigkeit eine große Rolle. Hier sind noch viele Spielräume offen und Optimierungen denkbar.

Ein Ansatz wäre es, zusätzliche Klassen von Cache-Groups und -Föderationen zu finden. Für diese Klassen könnten dann optimierte Modelle entwickelt werden. Denkbar sind hier z. B. eine Klasse von Cache-Groups, die lediglich homogene RCC-Zyklen enthalten. Außerdem könnten Cache-Groups bzw. -Föderationen in verschiedene „Regionen“ entsprechend Teilgraphen des SBG zerlegt werden, für die dann besondere Modelle angewendet werden können.

Eine weitere Verbesserung wäre eine mögliche Verknüpfung der verschiedenen Modelle. Für isolierte Wertemengen-Pfade in allgemeinen Cache-Groups können eventuell Algorithmen unter Zuhilfenahme von Histogrammen zur Verbesserung der Ergebnisgenauigkeit eingesetzt werden. Auf der anderen Seite ist auch ein genaueres Modell zur Analyse der nicht-baumartigen RCC-zyklenfreien Cache-Groups möglich.

Die in dieser Arbeit vorgestellten Modelle sind nur konzeptionell entwickelt, wurden aber noch nicht implementiert. Lediglich zur „Einarbeitung“ in die Thematik wurde ein Stellen-Token-System implementiert, mit dem „experimentell“ eine Average-Case-Analyse versucht wurde. Dieser Ansatz wurde aber nicht weiter verfolgt. Die in Java erstellten Quelltexte sind unter [Bay04a] herunter zu laden.

Durch eine konkrete Implementierung der Modelle wäre es auch möglich, eine Evaluierung für verschiedene Arten von Cache-Groups durchzuführen. Auf diese Art und Weise könnte die Güte der verschiedenen Modelle bewertet werden. Denkbar wären hier die Evaluierung des allgemeinen Modells anhand einer Cache-Group mit einem homogenen bzw. heterogenen Zyklus. Außerdem wäre eine Evaluierung des Modells mit Histogrammen für baumartige

Cache-Groups anhand einer solchen Cache-Group mit einer konkreten Verteilung der Werte in den Spalten möglich oder eine Bewertung des Modells für nicht-baumartige, RCC-zyklenfreie Cache-Groups.

Alles in Allem können wir sagen, dass für den Großteil an in der Praxis relevanten Cache-Groups Modelle zur Analyse des Füllstands gefunden wurden. Eine wichtige Rolle spielen in der Praxis z. B. baumartige Cache-Groups, denn sie unterstützen häufig verwendete Anfragetypen. Einige Systeme, z. B. die Datenbank-Caching-Lösung von TimesTen, unterstützen sogar nur diese Klasse von Cache-Groups.

Entsprechend der Bedeutung dieser Klasse von Cache-Groups haben wir ihr einen Großteil der Aufmerksamkeit in dieser Arbeit gewidmet. Die Algorithmen zur Berechnung der Füllstände in baumartigen Cache-Groups unter Zuhilfenahme von Histogrammen liefern die größte mögliche Genauigkeit der Ergebnisse.

Für allgemeine Cache-Groups und -Föderationen wurde zwar nur ein Modell für die Worst-Case-Analyse entwickelt, aber auch die Ergebnisse dieses Modells können einen guten Hinweis darauf geben, welche RCCs hilfreich sind und welche eventuell ein exzessives Füllverhalten nach sich ziehen können.

Anhang A: Literaturverzeichnis

[Wiki04a] Deutsche Wikipedia. Artikel über die hypergeometrische Verteilung. http://de.wikipedia.org/wiki/Hypergeometrische_Verteilung . Letzter Besuch Oktober 2004

[Wiki04b] Englische Wikipedia. Artikel über den Begriff „Cache“. http://en.wikipedia.org/wiki/Cache . Letzter Besuch Oktober 2004

[Wiki04c] Deutsche Wikipedia. Artikel über das Prädikat in der syllogistischen Logik. http://de.wikipedia.org/wiki/Pr%C3%A4dikat_%28Logik%29 . Letzter Besuch Oktober 2004

[Sun04a] Sun Microsystems. Produktbeschreibung zur JDBC-Technologie. http://java.sun.com/products/jdbc/ . Letzter Besuch Oktober 2004

[Sun04b] Sun Micosystems. Api-Beschreibung der prepared statements. http://java.sun.com/j2se/1.3/docs/api/java/sql/PreparedStatement.html Letzter Besuch Oktober 2004

[W3C04] World-Wide-Web-Consortium. Seiten des W3C über Web Services. http://www.w3c.org/2002/ws/ Letzter Besuch Oktober 2004

[Föl04] Helmut Föll. Beschreibung der verwendeten Stirling-Formel. http://www.tf.uni-kiel.de/matwis/amat/mw1_ge/kap_5/basics/m5_3_1.html . Letzter Besuch Oktober 2004

[IBM04] IBM Corporation. Homepage des DBCache-Projektes. http://www.almaden.ibm.com/software/dm/DBCache/index.shtml . Letzter Besuch Oktober 2004

[TT04] TimesTen Inc. Produktbeschreibung des Systems TimesTen/Cache. http://www.timesten.com/products/cache.html . Letzter Besuch Oktober 2004

[Bay04a] Christian Bayerlein. Quelltexte des experimentellen Token-Stellen-Systems. http://www.thalon.de/da/CacheSimulator.zip . Oktober 2004

[Bay04b] Christian Bayerlein. Quelltexte des Programms zur Evaluation der Modelle. http://www.thalon.de/da/eval.zip . Oktober 2004

[ABK+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh und Berthold Reinwald. Cache Tables: Paving theWay for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB 2003, Seiten 718–729. Morgan Kaufmann, 2003

[APT+03] Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. ICDE 2003: 821-831

[Ioa03] Yannis Ioannidis. The History of Histograms (abridged). VLDB 2003: 19-30. http://www.vldb.org/conf/2003/papers/S02P01.pdf

[Wei02] Gerhard Weikum. Web Caching. In: Web & Datenbanken – Konzepte, Architekturen, Anwendungen. Erhard Rahm / Gottfried Vossen (Hrsg.), dpunkt.verlag, 191-216 (2002)

[Här03] Theo Härder. unveröffentlichte Forschungsnotizen. 2003

[HB04a] Theo Härder und Andreas Bühmann. Database Caching – Towards a Cost Model for Populating Cache Groups. In *Advances in Databases and Information Systems, 8th East European Conference*, ADBIS 2004, Lecture Notes in Computer Science 3255, Seiten 215–229. Springer, 2004.

[HB04b] Theo Härder und Andreas Bühmann. Datenbank-Caching – Eine systematische Analyse möglicher Verfahren. *Informatik – Forschung und Entwicklung*, 19(1):2-16, Juli 2004.

[HB04c] Theo Härder und 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 . 2004.

[HB04d] Theo Härder und Andreas Bühmann. Query Processing in Constraint-Based Database Caches. http://wwwdvs.informatik.uni-kl.de/pubs/papers/HB04.DEB.pdf . *Data Engineering Bulletin*, 27(2):3-10, Juni 2004.

[Büh04] Andreas Bühmann. Einen Schritt zurück zum negativen Datenbank-Caching, 2004.

[LGZ04] Per-Åke Larson, Jonathan Goldstein und Jingren Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE 2004, Seiten 177–189. IEEE Computer Society, 2004.

[PB03] Stefan Podlipinig, Laszlo Böszörmenyi: A Survey of Web Cache Replacement Strategies. ACM Computing Surveys 35:4, 374-398 (2003)

Anhang B: Wichtige Schreibweisen und Symbolverzeichnis

An dieser Stelle sollen wichtige Schreibweisen für in dieser Arbeit benutzte Objekte aufgeführt werden. Dieser Anhang soll also nicht der inhaltlichen Klärung, sondern lediglich zum schnellen Nachschlagen von Schreibweisen dienen.

Zunächst bieten wir einen Überblick über die verschiedenen Schreibweisen an. Anschließend führen wir die verwendeten Symbole tabellarisch auf.

Cache-Groups, Spalten, Spaltenmengen, Cache-Tabellen und Mengen von Cache-Tabellen

Für eine (bestimmte, betrachtete) **Cache-Group** oder -**Föderationen** schreiben wir üblicherweise in Symbolen CG. **Cache-Tabellen** bezeichnen wir in dieser Arbeit normalerweise mit einem großen T, allgemeine **Spalten** mit c, **Konsumenten** mit k, **Erzeuger** mit e und Basiserzeuger mit b. Wenn mehrere Cache-Groups, -Föderationen, Cache-Tabellen oder Spalten zu unterscheiden sind verwenden wir einen Index.

Wir bezeichnen die **Menge der Cache-Tabellen** in einer Cache-Group oder -Föderation CG mit $\mathbf{CT}_{CG}$ und nennen die **Menge ihrer Spalten** $\mathbf{COL}_{CG}$. Wir verzichten auf die Angabe der Indizes, wenn die Cache-Group oder -Föderation implizit durch den Kontext bekannt ist.

Da jede Spalte zu einer Cache-Tabelle gehört, ist die **Cache-Tabelle-Zuordnungsfunktion** T(c) für jede Spalte $c \in \mathbf{COL}$ definiert.[31] Diese liefert zur Spalte c die Cache-Tabelle, zu der c gehört. Es gilt also:

$$T(c) = T_c \Leftrightarrow c \in T_c.$$

RCCs, Schmuggler-Kanten, Wertemengen-Pfade und weitere Spalten

Wir notieren **RCC-Kanten** (und allgemein **RCCs**) in dieser Arbeit mit $\rightarrow$ und **Schmuggler-Kanten** (und -**Beziehungen**) mit $\rightsquigarrow$. Wir schreiben also $e \rightarrow k$ für eine RCC von e dass k und $q \rightsquigarrow z$, wenn zwischen k und z eine Schmuggler-Beziehung besteht. Für einen **Wertemengen-Pfad**, der die Spalten $c_1,\ldots,c_i,\ldots,c_n$ verbindet, schreiben wir $c_1 \rightarrow \ldots \rightarrow c_i \rightarrow \ldots \rightarrow c_n$.

Wir schreiben K_T für die **Menge der Konsumenten** einer Cache-Tabelle T und für die **Menge ihrer Erzeuger** E_T. Falls eindeutig aus dem Kontext hervorgeht, zu welcher Cache-Tabelle die Konsumenten gehören, verzichten wir auf den Index. Für die **Menge der Nachfolger**

[31] Da die Funktion lediglich der einfacheren Schreibweise dient, wird sie in diesem Anhang aufgenommen. Die Funktion hat aber tatsächlich keinen Einfluß auf die Modelle.

eines Erzeugers e schreiben wir K_e. Umgekehrt schreiben wir E_k für die **Vorgänger** eines Konsumenten k.

Wir bezeichnen die **Menge der Basiserzeuger einer Spalte** c mit B_c und die **Menge der Basiserzeuger einer Cache-Tabelle** T mit B_T. Wir bezeichnen die Menge der **Basiserzeuger** e, **die dadurch nutzbar werden, dass b benutzt wurde** mit $B_n(b)$

Zeitpunkte

Für **Zeitpunkte** schreiben wir allgemein τ (eventuell erweitert durch ein Apostroph oder einen Index, um verschiedene Zeitpunkte unterschreiben zu können). Den (festen) Zeitpunkt vor einem betrachteten Füllzyklus bezeichnen wir mit α und den Zeitpunkt nach einem Füllzyklus (wenn die Cache-Group stabil ist) mit ω. Den frühesten Zeitpunkt, zu dem eine Cache-Tabelle T vollständig bevölkert ist, bezeichnen wir mit ω_T.

Für die **Menge aller Zeitpunkte**, zu denen **in einer Cache-Group oder -Föderation CG** ein Füllschritt ausgeführt wird, schreiben wir F_{CG}. Die **Menge aller Zeitpunkte**, bei der eine **Cache-Tabelle** T einen Füllschritt ausführt, wird mit F_T bezeichnet. Wir verzichten auch hier auf die Angabe der Indizes, wenn die Cache-Group oder -Föderation bzw. Cache-Tabelle implizit durch den Kontext bekannt ist.

Werte und Wertemengen

Wir bezeichnen einzelne **Werte** einer Wertemenge mit φ, wobei wir auch hier einen Index verwenden, um verschiedene Werte unterscheiden zu können. **Wertemengen** bezeichnen wir grundsätzlich mit einem großen V und einem entsprechenden Index. Die folgende Liste führt die wichtigsten Indizes auf, die wir in dieser Arbeit verwenden:

- Die **Menge aller Werte**, die in den DBDs relevanter Spalten einer Cache-Group bzw. -Föderation CG vorkommen bezeichnen wir mit V_{CG}.
- Außerdem führen wir die symbolische Bezeichnung V_c für die **DBD von Spalte c** ein.
- **Teilmengen der DBD einer Spalte** in einer stabilen Cache-Group benennen wir, indem wir den Index um ein entsprechendes Präfix *id* erweitern. Das Kürzel *id* bezeichnet eine Eigenschaft der in der Menge enthaltenen Werte. Wir verwenden Kürzel für umgebungsvollständige (vc), umgebungsfragmentarische (vf), geladene (l), induzierte, geschmuggelte (s), ungeladene (n) und referenzierte (ref) Werte. Der vertikale Strich im Index trennt dabei das Kürzel für die betrachtete Teilmenge von der Bezeichnung der Spalte.

  Beispiel: Die Menge der nach dem betrachteten Füllzyklus in einer Spalte c geladenen Werte bezeichnen wir so z. B. mit $V_{l|c}$.

- Betrachten wir Teilmengen der DBD einer Spalte **zu einem bestimmten Zeitpunkt**, so gibt es prinzipiell drei Möglichkeiten:
  - Wir können die Menge der Werte aus der DBD einer Spalte betrachten, die unmittelbar **vor dem Zeitpunkt** die Eigenschaft erfüllt haben (z. B. umgebungsvollständig, geladen, geschmuggelt etc. zu sein). In diesem Fall erweiterten wir den Index zusätzlich durch den betrachteten Zeitpunkt als Suffix, dass wiederum vom restlichen Index durch einen vertikalen Strich getrennt wird.

Beispiel: Die Menge der bis unmittelbar vor dem Zeitpunkt τ in einer Spalte c geladenen Werte bezeichnen wir so z. B. mit $V_{l|c|\tau}$.

- Wir können Werte betrachten, die eine bestimmte Eigenschaft zu einem Zeitpunkt τ erhalten, die also in die zugrunde liegende Wertemenge **zusätzlich hinzukommen**. In diesem Fall erweitern wir den Index ebenfalls mit einem Suffix aus einem vertikalen Strich und dem betrachteten Zeitpunkt, kennzeichnen aber V durch ein Apostroph. Außerdem notieren wir ebenfalls mit einem Apostroph die Wertemenge $V'_{g|c|\tau}$ der in die Spalte c zum Zeitpunkt τ generell zu ladenden Werte (d. h. zu diesem Zeitpunkt ankommende, in der DBD der verarbeitenden Spalte liegende Werte).

  Beispiel: Die Menge der zum Zeitpunkt τ in einer Spalte c zusätzlich geladenen Werte bezeichnen wir so z. B. mit $V'_{l|c|\tau}$. In dieser Menge liegen alle Werte, die vor dem Füllschritt zum Zeitpunkt τ noch nicht in c geladen waren, es aber zu einem Zeitpunkt τ' unmittelbar nach dem Füllschritt sind.[32]

- Für geschmuggelte Werte betrachten wir in dieser Arbeit außerdem die Menge der **während eines Füllschritts** zu einem Zeitpunkt τ in ein Schmuggelziel z geschmuggelte Werte. Für die Menge dieser Werte schreiben wir: $\tilde{V}_{s|z|\tau}$

- Eine Ausnahme stellt die Menge der über eine RCC induzierten Werte da. Hier wird der Eintrag, der üblicherweise die aktive Spalte bezeichnet, durch einen Eintrag für die betrachte RCC ersetzt.

  Beispiel: In Symbolen schreiben wir $V_{i|e\to k|\tau}$ für die Menge der bis zum Zeitpunkt τ **über e→k induzierten Werte** und $V'_{i|e\to k|\tau}$ für die Menge der während des Füllschritts zusätzlich induzierten Werte. Wie üblich schreiben wir $V_{i|e\to k}$ für die nach dem Füllzyklus insgesamt über e→k induzierten Werte.

- In analoger Weise notieren wir die Menge der **über einen Wertemengen-Pfad induzierten** Werte. Wir schreiben also $V_{i|c_1\to\ldots\to c_n}$ für die Menge der nach dem Füllzyklus über den Wertemengen-Pfad $c_1\to\ldots\to c_n$ induzierten Werte. Gleiches gilt natürlich ebenfalls für den Menge der bis zum Zeitpunkt τ über den Wertemengen-Pfad induzierten Werte (Schreibweise: $V_{i|c_1\to\ldots\to c_n|\tau}$) und für die zu diesem Zeitpunkt über den Wertemengen-Pfad zusätzlich induzierten Werte (Schreibweise: $V'_{i|c_1\to\ldots\to c_n|\tau}$).

- Außerdem bezeichnen wir die (**abgeschätzte**) **Mächtigkeit** dieser Wertemenge mit einem kleinen v (bzw v') und dem jeweiligen Index, z. B. $v_{l|c}$ für die geschätzte Anzahl der geladenen Werte in c.

Bemerkung: Man beachte, dass die Mächtigkeit der DBD einer Spalte gleich der Kardinalität dieser Spalte im Backend ist: $|V_c| = c_c$. Aufgrund der kompakteren Schreibweise verwenden wir c_c für die Mächtigkeit der DBD einer Spalte. Da wir allgemeine Spalten ebenfalls durch

[32] D. h. es gibt keinen Zeitpunkt, der früher als τ' und später als τ ist und bei dem durch die Cache-Tabelle von c ein Füllschritt ausgeführt wurde.

ein kleines c bezeichnen, kann es hier theoretisch zu Verwechslungen kommen. Aus dem jeweiligen Kontext ergibt sich deshalb jeweils das Gemeinte.

Für die **Überdeckung** der DBD einer Spalte e zur DBD einer Spalte k schreiben wir $ü_{k,e}$. Dabei existiert üblicherweise eine RCC e→k.

Wahrscheinlichkeiten

Für einzelne Werte können wir außerdem die Wahrscheinlichkeit berechnen, mit der er eine gewisse Eigenschaft erfüllt. Wir bezeichnen für einen Wert φ die Wahrscheinlichkeit (konsistent zur Notation der Indizes von Wertemengen) vor Zeitpunkt τ in einer Spalte c geladen (bzw. induziert, geschmuggelt, referenziert) zu sein mit $p_{l|c|\tau}(\varphi)$ (bzw. mit $p_{i|c|\tau}(\varphi)$, $p_{s|c|\tau}(\varphi)$, $p_{ref|c|\tau}(\varphi)$). Bei Betrachtungen des Zustands nach dem Füllzyklus schreiben wir für die jeweiligen Wahrscheinlichkeiten $p_{id|c}(\varphi)$, wobei das Kürzel wieder die Eigenschaft in der üblichen Schreibweise darstellt (l = geladen, i = induziert usw.)..

Für die **Schmuggelwahrscheinlichkeit** (d. h. die Wahrscheinlichkeit für einen Wert φ, während eines betrachteten Füllschritts zum Zeitpunkt τ in Schmuggelziel z geschmuggelt zu werden), verwenden wir eine weitere Notation: $w_z(h, s)$. Dabei ist $h:=h_z(\varphi)$ die Häufigkeit, mit der φ in z vorkommt und s die Mächtigkeit der Schmuggelumgebung von z zum Zeitpunkt τ. Besonderheit dieser Schreibweise ist es, dass deutlich wird, dass es sich um eine von h und s abhängige Funktion handelt. Ist h fest durch die Gleichverteilungsannahme für alle Werte aus der DBD von z, so schreiben wir $p'_{s|z|\tau}(\varphi)=:w_z(h,s)=:g_z(s)$. Wenn wir die Anzahl der in z geschmuggelten Werte unter der Gleichverteilungsannahme berechnen, schreiben wir $v_{s|z}:=f_z(s)=c_z(1-g_z(s))$.

Sätze und Satzmengen

Nachdem wir nun ausführlich Schreibweisen für Wertemengen und Wahrscheinlichkeiten betrachtet haben, widmen wir uns nun Schreibweisen für Sätze und Satzmengen.

Sätze notieren wir allgemein durch das Symbol η. Für eine **Satzmenge** verwenden wir in dieser Arbeit grundsätzlich das Symbol S. Wir spezifizieren eine bestimmte Satzmenge durch Anhängen eines Index wie bei Wertemengen. Für die geschätzte Mächtigkeit einer Satzmenge schreiben wir ein kleines s (mit gleichem Index und/oder Ornament wie bei der zugehörigen Menge).

Die **Menge aller Sätze (im Backend zu) einer Cache-Tabelle** T benennen wir mit S_T. Außerdem bezeichnen wir die **Werteumgebung eines Werts** φ in Spalte c mit $S_{c,\varphi}$. Die Mächtigkeit dieser Werteumgebung liefert ein **Histogramm**, für das wir h_c schreiben. Für die Menge der in eine Cache-Tabelle T während eines Füllschritts zum Zeitpunkt τ **zusätzlich geladenen Sätze** schreiben wir $S'_{T|\tau}$.

Wir bezeichnen die **Werteumgebung einer Wertemenge**, indem wir an das Symbol S den gleichen Index wie bei der betrachteten Wertemenge anhängen. Für die Werteumgebungen einer Menge von zusätzlich hinzukommen Werten fügen wir wie bei der zugrunde liegenden Wertemenge ein Apostroph an das große S an.

Beispiel: Die Menge der Sätze, die einen Wert haben, der in Spalte c zum Zeitpunkt τ geladenen ist, bezeichnen wir mit $S_{l|c|\tau}$ und für die Werteumgebungen der zu diesem Zeitpunkt in die Spalte zusätzlich geladenen Werte schreiben wir $S'_{l|c|\tau}$.

Außerdem bezeichnen wir die (**abgeschätzte**) **Mächtigkeit** einer Satzmenge mit einem kleinen s (bzw s') und dem jeweiligen Index, z. B. $s_{l|T|\tau}$ für die geschätzte Anzahl zum Zeitpunkt τ in die Cache-Tabelle T geladenen Sätze.

Für die Menge der in eine Cache-Tabelle T in einer stabilen Cache-Group geladenen Sätze schreiben wir $S_{l|T}$, und für ihre abgeschätzte Mächtigkeit schreiben wir n_T. Die Menge der insgesamt zum Zeitpunkt τ in T geladenen Sätze notieren wir mit $S_{l|T|\tau}$

Für die Schmuggelumgebung (d. h. die Menge der Sätze, die zu einem gegebenen Zeitpunkt τ zu einem Schmuggeln von Werten in ein gegebenes Schmuggelziel z beitragen) schreiben wir $\tilde{S}_{s|z|\tau}$.

Besuche und Kontexte

In Kapitel 0 betrachten wir Besuche und Kontexte. Wir verwenden für eine Besuchskette das Symbol $\mathfrak{B}$ und für Kontexte schreiben wir $\mathfrak{k}$. Einen Kontext-Füllstand notieren wir mit $v_{\mathfrak{k}}$ und die zugehörige Wertemenge mit $V_{\mathfrak{k}}$.

Tabellarische Übersicht über die verwendeten Symbole

Im Anschluss sind die in dieser Arbeit verwendeten Symbole in mehreren Tabellen zusammen gestellt. Die Übersicht soll zum Nachschlagen dienen.

Symbol	Bezeichnetes Objekt
c	Spalte *oder* Kardinalität einer Spalte (ergibt sich aus dem Kontext)
ck	Cache-Key
e	Erzeuger
b	Basiserzeuger
k	Konsument
h_c	Histogramm der Spalte c
$ü_{k,e}$	Überdeckung der DBD von Spalte e zur DBD von Spalte k.
$e \rightarrow k$	RCC(-Kante) zwischen Erzeuger e und Konsument k
$c_1 \rightarrow \ldots \rightarrow c_n$	Wertemengen-Pfad zwischen Spalte c_1 und Spalte c_n
F	Menge von Zeitpunkten. Um welche Zeitpunkte es sich handelt wird durch ein Index spezifiziert.
τ	Zeitpunkt
p	Wahrscheinlichkeit
$w_z(h,s)$	Schmuggelwahrscheinlichkeit eines Werts mit Häufigkeit h in Schmuggelziel z bei Mächtigkeit s der Schmuggelwahrscheinlichkeit
$g_z(s)$	$:=w_z(h,s)$ für h konstant
$f_z(s)$	$:=c_z(1-g_z(s))$ (Abkürzung)
CG	Cache-Group (oder -Föderation)
COL	Menge der relevanten Spalten einer Cache-Group (oder -Föderation)
CT	Menge der Cache-Tabellen einer Cache-Group (oder -Föderation)
$\overleftarrow{P}$	Menge maximaler Wertemengen-Pfade, die an einer bestimmten Spalte oder Cache-Tabelle enden. Um welcher Spalte bzw. Cache-Tabelle es sich handelt, wird durch einen Index spezifiziert.
$\vec{P}$	Menge von Wertemengen-Pfaden, die an einer bestimmten Spalte oder

	Cache-Tabelle anfangen. Um welche Spalte bzw. Cache-Tabelle es sich handelt, wird durch einen Index spezifiziert.
V und V’	Wertemenge. Um welche Werte es sich handelt wird durch ein Index spezifiziert.
$\tilde{V}_{s\|c\|\tau}$	Menge der Werte, die während des Füllschritts zum Zeitpunkt τ in Spalte c geschmuggelt wurden
φ	Wert
S	Satzmenge. Um welche Sätze es sich handelt wird durch ein Index spezifiziert.
S’	ebenfalls eine Satzmenge. Es handelt sich wieder um eine Menge neuer Sätze oder um die Werteumgebung einer Menge zusätzlich hinzugekommener Werte – je nach Index
$\tilde{S}_{s\|z\|\tau}$	Schmuggelumgebung von z zum Zeitpunkt τ
$\mathfrak{B}$	Besuchskette
$\mathfrak{k}$	Kontext
v, v’, s, s’…	Abgeschätzte Mächtigkeit der zugrunde liegenden Wertemenge oder Satzmenge, die durch einen Index spezifiziert wird.
$v_{\mathfrak{k}}$	Kontext-Füllstand
n	Füllstand einer Cache-Tabelle (welche Cache-Tabelle gemeint ist bestimmt ein Index)
T	Cache-Tabelle

Tabelle 4: Übersicht über die verwendeten Symbole

Index	In der Menge enthaltene Werte
CG	Alle Werte aus den DBDs aller relevanten Spalten der Cache-Groups CG
ohne Index	Alle im Rechner darstellbaren Werte
$\mathfrak{k}$	Alle Werte, die den Füllstand des Kontext $\mathfrak{k}$ beeinflussen.
c	Alle Werte aus der DBD der Spalte c
id|c	Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft nach dem Füllzyklus haben.
id|c|τ	Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft vor dem Zeitpunkt τ (τ *nicht* mit eingeschlossen) haben.
i|e→k	Werte, die nach dem Füllzyklus über die RCC e→k (in k) induziert sind.
i|e→k|τ	Werte, die vor Zeitpunkt τ über die RCC e→k (in k) induziert sind.
i|c_1→…→c_n	Werte, die nach dem Füllzyklus über den Wertemengen-Pfad c_1→…→c_n (in c_n) induziert sind.
i|c_1→…→c_n|τ	Werte, die vor Zeitpunkt τ über den Wertemengen-Pfad c_1→…→c_n (in c_n) induziert sind.v

Tabelle 5: Indizes von Wertemengen V (nicht V’) aller Werte mit einer bestimmten Eigenschaft

Index	In der Menge enthaltene Werte
id|c|τ	zum Zeitpunkt τ zusätzlich hinzugekommene Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft haben.

g|c|τ	alle Werte, die zum Zeitpunkt τ in Spalte c generell zu laden sind (d. h. dort ankommen und in der DBD von c liegen)
i|e→k|τ	Werte, die zum Zeitpunkt τ zusätzlich über die RCC e→k (in k) induziert wurden.
i|c_1→…→c_n|τ	Werte, die vor Zeitpunkt τ zusätzlich über den Wertemengen-Pfad c_1→…→c_n (in c_n) induziert wurden.

Tabelle 6: Indizes von Wertemengen V' der zusätzlichen mit einer Eigenschaft hinzugekommen Werte

Kürzel *id*	**Eigenschaft der Werte (die Werte sind…)**
vc	umgebungsvollständig
vf	umgebungsfragmentarisch
n	ungeladen
l	geladen
i	induziert
s	geschmuggelt
ref	ein durch eine Benutzeranfrage referenzierter Cache-Key-Wert

Tabelle 7: Bezeichner für Eigenschaften von Werten (Verwendung in Indizes von Wertemengen und Satzmengen)

Index	**In der Menge enthaltene Sätze**
T	Alle im Backend der Cache-Tabelle T enthaltenen Sätze
l|T	Sätze, die nach dem Füllzyklus in der Cache-Tabelle T geladen sind
l|T|τ	Sätze, die vor dem Zeitpunkt τ in der Cache-Tabelle T geladen sind
c,φ	Werteumgebung von Wert φ in Spalte c
id|c	Werteumgebung der Menge der Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft nach dem Füllzyklus haben.
id|c|τ	Werteumgebung der Menge der Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft vor dem Zeitpunkt τ (τ *nicht* mit eingeschlossen) haben.

Tabelle 8: Indizes von Satzmengen S (nicht S')

Index	**In der Menge enthaltene Sätze**
l|T|τ	Sätze, die zum Zeitpunkt τ in der Cache-Tabelle T geladen wurden.
id|c|τ	Werteumgebung der Menge der zum Zeitpunkt τ zusätzlich hinzugekommen Werte aus der DBD der Spalte c, die die mit dem Kürzel *id* bezeichnete Eigenschaft haben.

Tabelle 9: Indizes von Satzmengen S'

Anhang C: Beweise

In diesem Anhang sind ausführliche Beweise für Folgerungen zur Thematik der Schmuggelwahrscheinlichkeit zu finden. Der Umfang dieser Beweise würde den Rahmen des Hauptteils dieser Arbeit sprengen. Der Vollständigkeit halber seien die Beweise dennoch an dieser Stelle zusammen gestellt. Wir verwenden als Abkürzung für die Anzahl $|S_{T(z)}|$ der im Backend der Tabelle T(z) gespeicherten Sätze in diesem Anhang den Buchstaben N. Des Weiteren kürzen wir Häufigkeiten mit $h:=h_z(\varphi)$ und die Mächtigkeit der Schmuggelumgebung durch $s:=\tilde{s}_{s|z|\tau}$ ab.

Zunächst wollen wir eine Herleitung von Formel 2.8.1 zur Berechnung der Schmuggelwahrscheinlichkeit aufzeigen. Sei dazu z ein Schmuggelziel. Dann gilt:

$$w_z(h,s) = 1 - \frac{\binom{N-h}{s}}{\binom{N}{s}} = \frac{(N-h)!\cdot(N-s)!}{(N-h-s)!\cdot N!}. \qquad \textit{Formel 2.8.1}$$

Beweis: Wir modellieren die Schmuggelwahrscheinlichkeit durch ein Zufallsexperiment.

Wir ziehen aus einer Urne mit N Bällen. Unter diesen Bällen befinden sich h rote Bälle. Der Rest der Bälle ist weiß. Wir ziehen aus dieser Urne s Bälle.

Das Experiment modelliert die Auswahl der geschmuggelten Werte. Die Bälle sind also die Sätze im Backend und die Farbe repräsentiert die Eigenschaft, ob der Satz in der Werteumgebung von φ in z liegt. Das Ziehen der Bälle modelliert das Laden der Sätze in der Schmuggelumgebung aus dem Backend in die Cache-Tabelle. Ist einer der in der Schmuggelumgebung liegenden Sätze (also einer der gezogenen Bälle) in der Werteumgebung von φ in z (also rot), dann wurde der Wert in z geschmuggelt. Die Wahrscheinlichkeit, dass ein gezogener Ball rot ist, entspricht also der Wahrscheinlichkeit, dass ein Satz aus der Schmuggelumgebung in der Werteumgebung von φ in z liegt.

Sei A nun das Ereignis, dass die Ziehung der Bälle einen roten Ball enthält. Dieses Ereignis repräsentiert, dass der Wert φ in z geschmuggelt wurde. Wir können die Wahrscheinlichkeit dieses Ereignisses berechnen, indem wir das umgekehrte Ereignis $\overline{A}$, also dass keiner der der Bälle rot ist (bzw. der Wert nicht geschmuggelt wurde) betrachten:

$$p(A) = 1 - p(\overline{A}).$$

Wir schätzen die Wahrscheinlichkeit dieses Ereignisses ab, indem wir Ziehungen als Elementarereignisse mit gleichverteilter Wahrscheinlichkeit betrachten und die Anzahl der günstigen Ziehungen durch die Anzahl der möglichen Ziehungen teilen.

Der Wahrscheinlichkeitsraum Ω entspricht also der Menge der Elementarereignisse, d. h. der Menge der möglichen unterschiedlichen Ziehungen. Wir können die Mächtigkeit des Wahrscheinlichkeitsraums wie folgt berechnen:

$$|\Omega| = \binom{N}{s}.$$

Der Binomialkoeffizient berechnet die Anzahl der Teilmengen mit Mächtigkeit s aus einer gemeinsamen Obermenge mit Mächtigkeit N. Diese Teilmengen entsprechen den möglichen Ziehungen.

Die Anzahl $\overline{A}$ der günstigen Elementarereignisse, d. h. die Anzahl der Ziehungen, die keinen roten Ball enthalten, kann ebenfalls durch einen Binomialkoeffizient berechnet werden. Hier reduziert sich die Mächtigkeit der gemeinsamen Obermenge um die Anzahl der roten Bälle:

$$|\overline{A}| = \binom{N-h}{s}.$$

Die Wahrscheinlichkeit, dass keiner der gezogenen Bälle rot ist, kann nun abgeschätzt werden, indem die Anzahl der günstigen Elementarereignisse durch die Mächtigkeit des Wahrscheinlichkeitsraums geteilt wird. Es gilt dementsprechend:

$$p(\overline{A}) = |\overline{A}| / |\Omega| = \frac{\binom{N-h}{s}}{\binom{N}{s}}.$$

Daraus folgt:

$$w_z(h,s) = p(A) = 1 - p(\overline{A}) = 1 - |\overline{A}| / |\Omega| = 1 - \frac{\binom{N-h}{s}}{\binom{N}{s}}.$$

Binomialkoeffizienten können durch Fakultäten wie folgt dargestellt werden:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

Einsetzen und kürzen bringt:

$$1 - \frac{\binom{N-h}{s}}{\binom{N}{s}} = 1 - \frac{(N-h)!}{s! \cdot (N-h-s)!} \Big/ \frac{N!}{s! \cdot (N-s)!} = 1 - \frac{(N-h)! \cdot (N-s)!}{(N-h-s)! \cdot N!}. \blacksquare$$

Zu *Folgerung 14*: Die Schmuggelwahrscheinlichkeit kann mit Hilfe der Stirling-Formel wie folgt angenähert werden:

$$w_z(h, s) \approx 1 - \frac{(N-h) \cdot (N-s)}{(N-h-s) \cdot N}.$$

Beweis: Setzen wir die Stirling-Formel in die Gleichung ein, so gilt:

$$w_z(h, s) \approx 1 - \frac{e^{N-h} \cdot e^{N-s}}{e^{N-h-s} \cdot e^{N}} \cdot \frac{(N-h) \cdot (N-s)}{(N-h-s) \cdot N}.$$

Schauen wir uns den ersten Faktor genauer an:

$$\frac{e^{N-h} \cdot e^{N-s}}{e^{N-h-s} \cdot e^{N}} = e^{N-h+N-s-N+h+s-N} = e^{0} = 1.$$

Einsetzen in die erste Gleichung bringt:

$$w_z(h, s) \approx 1 - 1 \cdot \frac{(N-h) \cdot (N-s)}{(N-h-s) \cdot N} = 1 - \frac{(N-h) \cdot (N-s)}{(N-h-s) \cdot N}. \blacksquare$$

Zu *Folgerung 15*. Es gilt:

$$\overline{w}_z(h,1) = \frac{N-h}{N}, \qquad \overline{w}_z(h,s+1) = \overline{w}_z(h,s) \cdot \frac{N-h-s}{N-s}$$

Beweis: Wir zerlegen die Darstellung aus Formel 2.8.1 (siehe oben):

$$\overline{w}_z(h,s) = \frac{(N-h)!\cdot(N-s)!}{(N-h-s)!\cdot N!} = \frac{(N-h)!}{N!} \cdot \frac{(N-s)!}{(N-h-s)!}$$

Wir stellen die Fakultäten als Produktfolgen dar und erhalten:

$$\frac{(N-h)!}{N!} \cdot \frac{(N-s)!}{(N-h-s)!} = \frac{1 \cdot 2 \cdot \ldots \cdot (N-h)}{1 \cdot 2 \cdot \ldots \cdot N} \cdot \frac{1 \cdot 2 \cdot \ldots \cdot (N-s)}{1 \cdot 2 \cdot \ldots \cdot (N-h-s)}.$$

Betrachten wir den ersten Faktor. Da die Mächtigkeit der Werteumgebung und damit der Histogrammwert kleiner als die Anzahl der Sätze im Backend ist, gilt:

$$N - h \leq N.$$

Damit kürzt sich der Zähler gegen die ersten Faktoren des Nenners und es ergibt sich:

$$\frac{1 \cdot 2 \cdot \ldots \cdot (N-h)}{1 \cdot 2 \cdot \ldots \cdot N} = \frac{1}{(N-h+1) \cdot \ldots \cdot N}.$$

Schauen wir uns nun den rechten Faktor an. Wenn wir in der obigen Ungleichung auf beiden Seiten s abziehen, ergibt sich:

$$N - h - s \leq N - s.$$

In diesem Fall kürzt sich also der Nenner komplett weg und es bleibt stehen:

$$\frac{1 \cdot 2 \cdot \ldots \cdot (N-s)}{1 \cdot 2 \cdot \ldots \cdot (N-h-s)} = (N-h-s+1) \cdot \ldots \cdot (N-s).$$

Durch Einsetzen ergibt sich dann:

$$\frac{(N-h)!}{N!} \cdot \frac{(N-s)!}{(N-h-s)!} = \frac{(N-h-s+1) \cdot \ldots \cdot (N-s)}{(N-h+1) \cdot \ldots \cdot N}.$$

Den Anfang der rekursiven Darstellung erhält man durch einfaches Einsetzen:

$$\overline{w}_z(h,1) = \frac{(N-h-1+1)\cdot\ldots\cdot(N-1)}{(N-h+1)\cdot\ldots\cdot N} = \frac{(N-h)\cdot\ldots\cdot(N-1)}{(N-h+1)\cdot\ldots\cdot N} = \frac{N-h}{N}.$$

Schauen wir uns nun die zweite Gleichung an. Wir setzen ein und erhalten:

$$\overline{w}_z(h,s+1) = \frac{(N-h-(s+1)+1)\cdot(N-h-(s+1)+2)\cdot\ldots\cdot(N-h-(s+1)-1)\cdot(N-(s+1))}{(N-h+1)\cdot(N-h+2)\cdot\ldots\cdot(N-1)\cdot N} =$$

$$\frac{(N-h-s)\cdot(N-h-s+1)\cdot\ldots\cdot(N-h-s-2)\cdot(N-s-1)}{(N-h+1)\cdot(N-h+2)\cdot\ldots\cdot(N-1)\cdot N} =$$

$$\frac{(N-h-s+1)\cdot\ldots\cdot(N-s)}{(N-h+1)\cdot\ldots\cdot N}\cdot\frac{N-h-s}{N-s} = \overline{w}_z(h,s)\cdot\frac{N-h-s}{N-s}. \blacksquare$$

Zu *Folgerung 16*: Sei $s \geq 0$ und

$$\tilde{w}_z(h,s) = \overline{w}_z(h,\lfloor s\rfloor)\cdot\left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{s-\lfloor s\rfloor}.$$

Dann gilt:

$$\tilde{w}_z(h,s) = \overline{w}_z(h,s) \text{ für ganzzahlige s und } \overline{w}_z(h,\lfloor s\rfloor) \geq \tilde{w}_z(h,s) \geq \overline{w}_z(h,\lceil s\rceil).$$

Beweis: Sei zunächst s ganzzahlig. Dann gilt:

$$\tilde{w}_z(h,s) = \overline{w}_z(h,s)\cdot\left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^0 = \overline{w}_z(h,s)\cdot 1 = \overline{w}_z(h,s).$$

Sei nun s reell, dann müssen wir zwei Fälle unterschreiben:

Fall 1) Zunächst gelte $s > N - h + 1$. Dann gilt

$$0 = \overline{w}_z(h,\lfloor s\rfloor) \geq \tilde{w}_z(h,s) \geq \overline{w}_z(h,\lceil s\rceil) = 0.$$

Die Gleichung wird null, denn bei der rekursiven Formulierung stoßen wir irgendwann auf einen Faktor

$$\overline{w}_z(h,N-h+1) = \overline{w}_z(h,N-h)\cdot\frac{N-h-N+h}{N-N+h} = 0,$$

und damit werden automatisch alle $\overline{w}_z(h,\lfloor s\rfloor)$ mit $s > N - h + 1$ gleich null.

Fall 2) Sei nun also $s \leq N - h + 1$, dann gilt:

$$0 \leq \frac{N-h-s}{N-s} \leq 1.$$

Auf der anderen Seite gilt

$$\left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{s-\lfloor s\rfloor} < \left(\frac{|S_{T(z)}|-h-\lceil s\rceil}{|S_{T(z)}|-\lceil s\rceil}\right)^{0} = 1$$

und daher $\overline{w}_z(h,\lfloor s\rfloor) \geq \overline{w}_z(h,\lfloor s\rfloor) \cdot \left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{s-\lfloor s\rfloor} = \tilde{w}_z(h,s)$.

Außerdem können wir schreiben:

$$\overline{w}_z(h,\lceil s\rceil) = \overline{w}_z(h,\lfloor s\rfloor+1) = \overline{w}_z(h,\lfloor s\rfloor) \cdot \frac{N-h-(\lfloor s\rfloor+1)}{N-(\lfloor s\rfloor+1)}$$

und wegen $\lfloor s\rfloor+1 = \lceil s\rceil$ gilt:

$$\overline{w}_z(h,\lceil s\rceil) = \overline{w}_z(h,\lfloor s\rfloor) \cdot \frac{N-h-\lceil s\rceil}{N-\lceil s\rceil} = \overline{w}_z(h,\lfloor s\rfloor) \cdot \left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{1}$$

woraus folgt:

$$\tilde{w}_z(h,s) = \overline{w}_z(h,\lfloor s\rfloor) \cdot \left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{s-\lfloor s\rfloor} \geq \overline{w}_z(h,\lfloor s\rfloor) \cdot \left(\frac{N-h-\lceil s\rceil}{N-\lceil s\rceil}\right)^{1} = \overline{w}_z(h,\lceil s\rceil).$$

Womit beide Schranken der Behauptung gelten. ■

Zu *Folgerung 17*: Sei z ein Schmuggelziel und s die abgeschätzte Mächtigkeit der Schmuggelumgebung in z. Dann gilt:

$$v_{s|z} = \sum_{\varphi\in V_z} w_z(h_z(\varphi),s).$$

Beweis: Sei I_φ eine Zufallsvariable, die angibt, ob φ in z geschmuggelt wurde. Formal sei

$$I_\varphi = \begin{cases} 1 \text{ wenn } \varphi\in V_{l|s} \\ 0 \text{ sonst} \end{cases}$$

Das zugehörige Ereignis sei A_φ. Wir können die Wahrscheinlichkeit dieses Ereignisses bereits berechnen:

$$p(A_\varphi) = p_{s|z}(\varphi) = w_z(h_z(\varphi), s_{s|z}).$$

Für den Erwartungswert von I_φ, $E(I_\varphi)$, gilt also:

$$E(I_\varphi) = 1\cdot p(A_\varphi) + 0\cdot p(\overline{A_\varphi}) = p(A_\varphi).$$

Dann gilt $v_{s|z} = \sum_{\varphi\in V_e} I_\varphi$ und somit:

$$v_{s|z} = E(\sum_{\varphi\in V_e} I_\varphi) = \sum_{\varphi\in V_e} E(I_\varphi) = \sum_{\varphi\in V_e} p(A_\varphi) = \sum_{\varphi\in V_z} w_z(h_z(\varphi),s).$$ ■

Folgerung 18: Sei z ein Schmuggelziel und s die abgeschätzte Mächtigkeit der Schmuggelumgebung von z. Des Weiteren gelte die Gleichverteilungsannahme. Dann kann $v_{s|z}$ folgendermaßen berechnet werden:

$$v_{s|z} = c_z \cdot w_z(N / c_z, s) .$$

Beweis: Seien I_φ und A_φ wie im Beweis zur *Folgerung 17*. Dann gilt:

$$p(A_\varphi) = w_z(h_z(\varphi), s) = w_z(N / c_z, s) \text{ für alle } \varphi \in V_z.$$

Und da die Schmuggelwahrscheinlichkeit für jeden Wert gleich groß ist, können wir die Summe aus *Folgerung 17* durch ein Produkt ersetzen. Wir haben c_z Summanden, also gilt:

$$v_{s|z} = \sum_{\varphi \in V_z} w_z(N / c_z, s) = c_z \cdot w_z(N / c_z, s) . \blacksquare$$