Universität Kaiserslautern
SoSe 2004
Lehrgebiet Datenverwaltungssysteme
AG Datenbanken und Informationssysteme

Grundlagen webbasierter Informationssysteme

DB-Caching

Henning Klaßen
h_klasse@informatik.uni-kl.de

Juli 2004
Betreut von Andreas Bühmann

Inhaltsverzeichnis

1 Einleitung 3

2 Statische Realisierungen 3
2.1 Materialisierte Sichten . . . 4
2.1.1 Anfragen auf materialisierte Sichten . . . 4
2.1.2 Wartung von materialisierten Sichten . . . 6
2.2 Full-Table-Caching . . . 7

3 Dynamische Realisierungen 7
3.1 Replikationsfreie Sichten am Beispiel von DBProxy . . . 7
3.1.1 Cache Repository . . . 9
3.1.2 Anfragen-Matching-Modul . . . 10
3.1.3 Konsistenz-Manager . . . 11
3.1.4 Garbage Collection im Cache . . . 11
3.2 Cache Groups am Beispiel von DBCache . . . 12
3.2.1 Cache-Tabellen . . . 13
3.2.2 Cache Constraints . . . 13
3.2.3 Anfragen in DBCache . . . 16
3.2.4 Wartung von Cache-Tabellen . . . 16

4 Vergleich 18

5 Fazit 19

Literaturverzeichnis 20

1 Einleitung

Im Internet erlangen dynamische Seiten, also Seiten, die an den Benutzer individuell angepasst sind, wie z. B. von Online-Warenhäuseren wie Amazon, Ebay, eine immer größere Bedeutung. Es wird eine immer stärkere Benutzerorientierung angestrebt, die – aufgrund der dynamisch generierten Inhalte – eine Datenbank benötigt, in der alle Daten abrufbar sind. Für solche Seiten ist das Web-Caching [Ada04] nur schlecht geeignet, da Zugriffe auf Datenbanken stattfinden, deren Inhalte durch Web-Caching nicht auf einem so genannten „Edge-“ oder „Front-End-Server“ (FE-Server) gespeichert werden können. Zugriffe von mehreren Benutzern gleichzeitig auf den selben Back-End-Server (BE-Server), der all diese Fragen beantworten soll, resultieren allerdings in hohen Geschwindigkeitseinbußen – und welcher Kunde will schon mehrere Sekunden warten, bis sich eine Seite aufgebaut hat? Aus diesem Grund wird Datenbank-Caching (DB-Caching) betrieben, welches die Daten des BE-Datenbankservers (BE-DBS) in eine lokale, autonome Datenbank auf dem FE-Server cacht. So kann die Last auf mehrere lokale FE-Server verteilt werden, die mit den vom Client benötigten Daten des dahinter liegenden BE-Servers versorgt werden [HB04b]. Natürlich sollten hierbei die Daten im Cache gespeichert sein, die von den Benutzern besonders oft abgefragt werden.

Die Herausforderung beim DB-Caching ist es, dass die Tabellen auf dem FE-Server die gleichen Ergebnisse auf eine Anfrage liefern wie die Tabellen auf dem BE-Server – unter Vernachlässigung der Zeitdifferenz für Update-, Delete- und Insert-Operationen auf dem BE und auf dem FE. Diese Anforderung an den Cache wird als Prinzip der *Konsistenz* bezeichnet. Um diese Anforderung zu gewährleisten, ist es notwendig, den Cache konsistent mit den jeweiligen Tabellen auf dem BE-Server zu halten. Somit müssen alle Änderungsoperationen, die am BE-DBS durchgeführt werden ebenso an den entsprechenden FE-Tabellen durchgeführt werden.

Außerdem ist bei einer Anfrage zu beachten, dass die *Vollständigkeit* gewährleistet ist. Das bedeutet, dass sich die Tupel, welche die Anfrageprädikate erfüllen, komplett im Cache befinden. Es wird jedoch nicht gefordert, dass alle Prädikate im Cache auswertbar sein müssen. Es reicht, wenn nur eine Teilmenge von Prädikaten der Anfrage im Cache beantwortet werden können. Die Prädikate, die im Cache beantwortet werden können, müssen allerdings die oben beschriebene Konsistenz und Vollständigkeit erfüllen. Die übrigen Prädikate können an den BE-DBS weitergeleitet und dort ausgewertet werden. Die Ergebnisse aus den Cache-Tabellen und den Originaltabellen auf dem BE-Server werden dann zum Anfrageergebnis zusammengebaut und anschließend dem Benutzer übermittelt.

Ein weiteres wichtiges Prinzip ist die *Transparenz* des Cachings für den Entwickler einer Applikation. Der Entwickler soll von der hinter dem Caching steckenden Logik bei seinen Anfragen nichts berücksichtigen müssen, sondern wie gewohnt die Anfragen an die BE-Tabellen stellen, die dann – für den Entwickler unsichtbar – auf den Cache umgeleitet werden.

In diesem Seminar sollen Strategien zum DB-Caching behandelt werden mit besonderer Rücksicht darauf, wie diese die oben genannten Anforderungen erfüllen. Dabei wird unterschieden zwischen den statischen Ansätzen, die den Inhalt der Datenbank nicht dynamisch anhand der Anfragelast modifizieren, und den dynamischen, welche durch verschiedene Mechanismen eine dynamische Anpassung des Caches an die Anfragelast gewährleisten.

2 Statische Realisierungen

In diesem Abschnitt werden die statischen Realisierungen behandelt, welche die Cache-Tabellen des FE-Servers – wie ihr Name schon sagt – nicht dynamisch anhand der Anfragelast geeignet anpassen, also die häufig vom Benutzer geforderte Inhalte nicht direkt in den Cache laden. Der DBA muss also die Inhalte der Tabellen auf den FE-Servern entsprechend nachbessern. Allerdings sind die im Folgenden beschriebenen Techniken sehr einfach und erfordern keine zusätzlichen Erweiterungen des SQL-Befehlssatzes oder ähnliches. Recht häufig verwendete Techniken zum statischen DB-Caching sind die materialisierten Sichten und die volle Replikation aller Daten der BE-Tabellen.

Anfrage		Sicht	
SELECT	Abt.Name, Pers.Name,	SELECT	Abt.Name, Pers.Name,
	Gehalt, Alter, AbtCluster.Umsatz		Gehalt, Alter, AbtCluster.Umsatz
FROM	Pers, Abt, AbtCluster	FROM	Pers, Abt, AbtCluster
WHERE	Gehalt >10000	WHERE	Gehalt >5000
AND	Abt.Id = Pers.AbtId	AND	Abt.Id = Pers.AbtId
AND	Abt.Id = AbtCluster.AbtId	AND	Pers.AbtId = AbtCluster.AbtId
AND	Pers.Beruf = 'Programmierer'	AND	Pers.Beruf = 'Programmierer'
AND	AbtClust.Name LIKE '%PC%'		

Abbildung 1: Beispiel für eine Sicht und eine Anfrage

2.1 Materialisierte Sichten

Eine einfache Art, DB-Caching zu implementieren, – abgesehen von der vollen Replikation – stellen sicherlich die materialisierten Sichten (MSen) dar. Das Konzept ist recht simpel: Auf dem FE-Server werden die Cache-Tabellen als materialisierte Sichten vom DBA definiert. Es ist am günstigsten, wenn auf dem jeweiligen FE-Server solche Tupel in materialisierten Sichten gespeichert werden, auf die die Benutzer besonders häufig zugreifen werden.

Weil das Konzept der materialisierten Sichten recht verbreitet ist und es keine spezielle Lösung für DB-Caching-Verfahren darstellt, wird im Folgenden nur kurz darauf eingegangen.

2.1.1 Anfragen auf materialisierte Sichten

Das wohl Wichtigste bei MSen ist die Prüfung, ob eine Anfrage von der Sicht beantwortet werden kann oder nicht, also ob die *Vollständigkeit* der Tupel im Cache für die Anfrage gegeben ist. Gerade beim DB-Caching spielt dies, wie oben dargelegt, eine herausragende Rolle. Zu überprüfen ist zum einen, ob eine MS alle Tupel besitzt, die auch in der Anfrage vorkommen, und ob die Spalten, die in den Tupeln vorkommen, auch das gleiche Intervall von Werten haben wie die Spalten in der Anfrage. Bei einer Anfrage über alle Personen, die zwischen 20 und 30 Jahren alt sind, kann z. B. keine Sicht verwendet werden, die nur alle Personen im Alter von 20 bis 25 Jahren enthält.

Es werden im Folgenden nur Anfragen und Sichten betrachtet, die sich auf die gleichen Tabellen beziehen. Erweiterungen dieses Konzepts auf mehrere Tabellen und auf Aggregationen sind möglich, werden hier aber nicht behandelt [GL01].

Wenn eine Sicht ausreicht, eine Anfrage zu beantworten, so kann die Anfrage – gegebenenfalls mit gewissen Änderungen, wie z. B. dem Hinzufügen vom kompensierenden Prädikaten, welche weiter unten beschrieben werden – an die Sicht gestellt werden statt an die BE-Tabelle. Als Beispiel, wie dies überprüft werden kann, wird im folgenden der Algorithmus dargestellt, der im „Microsoft SQL Server"-Produkt verwendet wird [GL01].

Damit ein Vergleich möglich ist, müssen sich sowohl Sicht-Definition als auch Anfrage in KNF (konjunktiver Normalform) befinden. Die Prädikate werden dann aufgeteilt in Gleichheitsprädikate, Bereichsprädikate und die übrig bleibenden Prädikate. Als Beispiel hierzu siehe Abbildung 1. Hier sind z. B. Gleichheitsprädikate Abt.Id = Pers.AbtId und Abt.Id = AbtCluster.AbtId. Bereichsprädikate sind Gehalt > 10000, Gehalt > 5000 und Pers.Beruf = 'Programmierer'. Das übrig bleibende Prädikat ist AbtCluster.Name LIKE '%PC%'.

Zuerst werden die Gleichheitsprädikate in Äquivalenzklassen zusammengefasst. In einer Äquivalenzklasse (ÄK) sind alle Spalten enthalten, die gleich sind – sei dies nun direkt oder transitiv. Im Falle von (Abt.Id = Pers.AbtId AND Abt.Id = AbtCluster.AbtId) sind also die Spalten Abt.Id, Pers.AbtId und AbtCluster.AbtId in einer Äquivalenzklasse. Dadurch wird auch die Äquivalenz dieses Prädikates zu dem Prädikat (Abt.Id = Pers.AbtId AND Pers.AbtId = AbtCluster.AbtId) berücksichtigt. Somit existieren, nachdem der Algorithmus zur Bestimmung der ÄKn terminiert, nur noch Mengen, in denen gleiche Spalten enthalten sind oder sog. triviale ÄKn, die nur eine Spalte

als Element besitzen.

Im **Equijoin-Subsumptionstest** werden die nicht trivialen Äquivalenzklassen für die Anfrage und die Sicht verglichen. Sind alle nicht trivialen ÄKn der Sicht eine Untermenge der ÄKn der Anfrage, so gilt der Test als bestanden, da in diesem Fall alle Spalten, die in der Anfrage vorkommen auch in der Sicht enthalten sind. Hat die Sicht den Test nicht bestanden, so wird sie verworfen, da sie nicht alle Spalten besitzt, die die Anfrage benötigt und sie somit nicht benutzt werden kann, um die Anfrage zu beantworten. Falls es ÄKn der Anfrage gibt, die echte Obermengen von ÄKn der Sicht sind, so müssen kompensierende Prädikate erstellt werden. Ist z. B. $ÄK_a = \{C_a, C_b\}$ eine ÄK der Anfrage und $ÄK_s = \{C_a\}$ eine ÄK der Sicht, so muss das kompensierende Prädikat $C_a = C_b$ erstellt werden. Diese kompensierenden Prädikate werden, sobald die Sicht alle Tests bestanden hat und die Anfrage somit auf die Sicht umgeschrieben werden kann, dazu benötigt, die Sicht so einzuschränken, dass diese die erwünschten Ergebnisse liefert.

Beim **Bereichssubsumptionstest** wird überprüft, ob die durch die Where-Klausel festgelegten Wertebereiche der Spalten einer Sicht größer bzw. gleich den Wertebereichen der Anfrage sind. Ist dies nicht der Fall, so enthält die Sicht nur eine Teilmenge der von der Anfrage angeforderten Daten und ist somit zu deren Beantwortung ungeeignet. Gibt es ÄKn der Sicht, deren Wertebereich echt größer sind als die der ÄKn der Anfrage, so müssen entsprechende kompensierende Prädikate erstellt werden. Wenn wie in Abbildung 1 die Spalte Gehalt in einer ÄK der Sicht echt größer als 5000 ist, aber in der Anfrage echt größer als 10000 ist, so muss ein Prädikat erstellt werden, das den Bereich auf „echt größer 10000“ setzt.

Der **Subsumptionstest für verbleibende Prädikate** schließlich testet, ob die verbleibenden Prädikate – also all jene, die weder Gleichheitsprädikate sind noch das Wertintervall mit den Operatoren $\leq,\geq,<,>,=$ und einer Konstanten einschränken –, die in der Sicht vorkommen, mit einem Prädikat in der Anfrage übereinstimmen. Ist dies der Fall, so besteht die Sicht auch diesen Test, womit sichergestellt ist, dass auch die übrigen Prädikate in der Sicht vorhanden sind und diese somit zur Beantwortung der Anfrage geeignet ist. Gibt es von diesen verbleibenden Prädikaten der Anfrage allerdings welche, die nicht in der Sicht vorkommen, so müssen entsprechende kompensierende Prädikate erschaffen werden. In Abbildung 1 wäre dies das Prädikat AbtClust.Name LIKE '%PC%', das als kompensierendes Prädikat erschaffen werden müsste.

Bevor die Anfrage auf die Sicht umgeschrieben werden kann, muss noch überprüft werden, ob die Spalten in den kompensierenden Prädikaten auch in der Sicht vorhanden sind und ob alle Ausgaben der Anfrage auch von der Sicht berechnet werden können.

Um zu testen, ob alle Spalten vorhanden sind, wird für die Spalten aus kompensierenden Prädikaten, die im Equijoin- und im Bereichssubsumptionstest erstellt wurden, getestet, ob es mindestens eine Spalte in der Sicht gibt, die äquivalent zu einer Spalte aus der Anfrage ist. Für Spalten aus kompensierenden Prädikaten, die im letzten Subsumptionstest erstellt wurden, wird überprüft, ob jede Spalte auch in der Sicht vorkommt.

Um festzustellen, ob eine Sicht alle Ausgaben der Sicht berechnen kann, muss geprüft werden, ob alle zusätzlichen Prädikate von der Sicht korrekt berechnet werden können. Auch in diesem Schritt werden wieder die ÄKn zum Test herangezogen. Denn um zu bestimmen, ob eine Ausgabe berechnet werden kann, muss auch überprüft werden, ob es Ausdrücke mit anderen Spalten gibt, die das gleiche Ergebnis liefern.

Besteht die Sicht auch noch die beiden letzten Tests, so wird die alte Anfrage verworfen und eine neue Anfrage formuliert, welche in der FROM-Klausel die Sicht enthält und deren WHERE-Klausel eine Und-Verknüpfung der oben erstellten kompensierenden Prädikate ist.

Dadurch, dass der Algorithmus selbst überprüft, ob eine Anfrage lokal oder vom BE-DBS beantwortet werden kann, muss der Entwickler einer Applikation sich nicht darum kümmern, wohin er seine Anfragen richtet. Damit ist auch die *Transparenz* für dieses Verfahren gegeben.

Anfragen auf Sichten, die mehr Tabellen beinhalten als die Anfrage, bzw. Anfragen mit Aggregationen werden ebenfalls mit dem oben beschriebenen Algorithmus überprüft [GL01].

2.1.2 Wartung von materialisierten Sichten

Werden an Tabellen, auf denen ein oder mehrere materialisierte Sichten definiert sind, Update-, Delete- und Insert-Operationen (UDIs) ausgeführt, so müssen die entsprechenden MSen ebenfalls aktualisiert werden, damit die *Konsistenz* auch bei künftigen Anfragen auf den Cache gegeben bleibt. In einem naiven Ansatz könnte die Sichtendefinition einfach neu ausgewertet werden, was aber einen erheblichen Overhead mit sich bringen würde, da alle Tupel vom BE-DBS auf die jeweiligen FE-DBS geladen werden müssten. Günstiger ist es daher, nur die geänderten bzw. neuen Tupel in die Sicht einzufügen. Im Folgenden wird der „memoryless refresh"-Algorithmus von Oracle als Beispiel für die Wartung von MSen betrachtet, der einen Mechanismus zu der oben beschriebenen effektiven Wartung von MSen liefert.

Oracle protokolliert im *Zeilen-DML-Log*, welche Zeilen einer Tabelle via DML verändert wurden. Dazu werden in dem Log eine Kopie der geänderten Zeilen, deren Zeilen-Id, der DML-Typ (Update, Delete oder Insert) und eine Zeitmarke gespeichert. Die Zeitmarke dient beim Aktualisieren der MV dazu, festzustellen, ob diese Zeile in der MS aktualisiert bzw. hinzugefügt werden muss.

Um zu veranschaulichen, wie der Algorithmus arbeitet, gehen wir von zwei Tabellen A und B aus, auf denen durch einen natürlichen Join eine MS M definiert ist, also $M = A \bowtie B$. Bei diesem Join bleiben in M die Id-Spalten von A und B erhalten. Es ist interessanter, die Wartung an einer Sicht über mehrere Tabellen, die mit einem Join verbunden wurden, zu betrachten, da, falls die Sicht nur eine Tabelle enthält, die UDIs einfach an die Sicht weitergeleitet werden können. Seien weiter ΔA bzw. ΔB die Änderungen, die an A bzw. B durchgeführt wurden und seien A' und B' die geänderten Tabellen, also z. B. $A' = A \cup \Delta A$. Was es aber nun zu bestimmen gilt, sind die Änderungen, die an der Sicht durchgeführt werden müssen. Diese werden im Folgenden ΔM genannt. Um ΔM zu bestimmen, muss ΔA mit Tabelle B gejoint werden, ΔB entsprechend mit A und schließlich ΔA mit ΔB. Somit sind alle Kombinationen von Veränderungen in beiden Tabellen berücksichtigt. Dies führt zu folgender Gleichung [BDD+98]:

$$\begin{aligned} \Delta M &= (A \bowtie \Delta B) \cup (\Delta A \bowtie B) \cup (\Delta A \bowtie \Delta B) \\ &= (\Delta A \bowtie B') \cup (A \bowtie \Delta B) \end{aligned} \tag{1}$$

Da nach der Aktualisierung einer Tabelle die unveränderte Tabellen (in diesem Falle A und B) nur unter gewissen Zeitkosten zu berechnen sein können, lässt sich die Gleichung so umformen, dass sie keine ursprünglichen Tabellen mehr enthält [BDD+98]:

$$\Delta M = ((A' \bowtie \Delta B) \cup (\Delta A \bowtie B')) \setminus (\Delta A \bowtie \Delta B) \tag{2}$$

ΔB wiederum besteht aus drei Mengen: $\Delta B = U \cup D \cup I$. In den Mengen werden Zeilen gespeichert, die aktualisiert, gelöscht oder eingefügt werden sollen. Da der Algorithmus in vielen Situationen Update-Operationen durch eine Löschung gefolgt von einer Einfügeoperation behandelt, sind diese Zeilen entsprechend auch in den Mengen D und I enthalten. In diesem Fall arbeitet der Algorithmus in zwei Phasen:

1. *Löschphase:* Alle Zeilen in M, deren Zeilen-Id aus B in D enthalten sind, werden gelöscht.

2. *Einfügephase:* Das Ergebnis der Joins $A' \bowtie \Delta B$ wird für alle Elemente in I in M eingefügt.

Dieser Algorithmus wird zuerst für die Änderungen an Tabelle B, danach für Änderungen an der Tabelle A ausgeführt. In der Löschphase für Tabelle A werden die Tupel aus $\Delta A \bowtie \Delta B$ wieder gelöscht. Dies entspricht dem Ausschnitt „$\setminus (\Delta A \bowtie \Delta B)$" in Gleichung 2.

Der Algorithmus wird sowohl für MSen mit Joins, als auch für MSen mit Aggregationen und MSen mit Unteranfragen verwendet [BDD+98], was hier jedoch nicht genauer beschrieben wird.

2.2 Full-Table-Caching

Das Full-Table-Caching ist ein Extremfall des Ansatzes der materialisierten Sichten. Beim Full-Table-Caching wird der komplette Inhalt der BE-DB in die FE-DB kopiert. Es findet also eine Replikation der Daten statt. Der Vorteil liegt darin, dass keine komplizierten Mechanismen nötig sind, um zu entscheiden, ob eine Anfrage lokal ausgewertet werden kann oder ob sie an den BE-Server weitergereicht werden muss. Es muss somit auch nicht überprüft werden, ob alle Daten in der FE-DB vorhanden sind. Damit ist die *Vollständigkeit* gegeben, da sich der Cache nicht vom BE-DBS unterscheidet.

Der Nachteil dieses Verfahrens liegt auf der Hand. Zum einen verursacht volle Replikation einen hohen Speicheraufwand auf den FE-Servern, die meist von der Leistung her schwächer sind als die BE-Server. Außerdem besteht ein weiteres Problem darin, dass – um die Daten auf dem FE-Server aktuell zu halten und somit *Konsistenz* zu gewährleisten – in gewissen Abständen die FE-DB aktualisiert werden muss, was sich erheblich auf die Geschwindigkeit auswirkt [ABK+03]. Zudem sind die Daten vor der Aktualisierung veraltet, was für Server, die stets recht aktuelle Daten benötigen, nicht tolerabel ist. Dieses Problem tritt zwar auch bei den anderen beschriebenen Verfahren auf, ist dort aber nicht so wichtig wie hier, da dynamisch entschieden wird, welche Änderungen wichtig sind und somit möglichst bald an den Cache zu schicken sind. Bei MSen muss dagegen ein Mittelweg zwischen hoher Aktualität der Daten und geringer Auslastung des Netzes gefunden werden. Werden nämlich jede Nacht die FE-Server aktualisiert, so ist die Last gering und der Benutzer bekommt davon wenig mit. Dafür wird die Aktualität der Daten um so schlechter, je weiter der Tag voranschreitet.

Da Full-Table-Caching ein Extremfall von MSen ist, ist auch das in Abschnitt 2.1.1 beschriebene Verfahren verwendbar. Das gewährleistet die *Transparenz* des Caches für den Programmierer.

3 Dynamische Realisierungen

Im Gegensatz zu den statischen Realisierungen zielen die dynamischen Realisierungen darauf ab, dass das Verfahren selbst anhand der Last entscheidet, welche Anfragen lokal, d. h. im FE gespeichert werden sollen. Je nach Verfahren werden die Anfrageergebnisse und – implizit oder explizit – die Anfrage selbst im Cache gespeichert, was bei den einzelnen Verfahren in den Abschnitten 3.1 und 3.2 genauer beschrieben wird. Die Verfahren sind also adaptiv, was für den Administrator bedeutet, dass er außer der Entscheidung, welche Tabellen und Tupel initial im FE gespeichert werden sollen, und der Initialisierung der FE-DBS keine weitere Arbeit hat. Alle späteren Anpassungen der FE-Tabellen wie z. B. UDIs übernimmt das Verfahren. Im Folgenden werden zwei dieser Verfahren beschrieben, nämlich DBProxy, welches je nach Anfragelast ggf. nur einen Teil des Schemas einer Tabelle im Cache speichert, und DBCache, das vorab eine Spezifikation der im Cache vorhandenen Tabellen erwartet, also das komplette Tabellenschema übernimmt. Das Semantic Caching, das allerdings nicht so weit ausgereift ist wie DBCache und DBProxy und daher hier ausgelassen wird, wählt einen ähnlichen Ansatz wie DBProxy. So wird der Cache-Inhalt durch Prädikate beschrieben, die zur Überprüfung, ob eine Anfrage den Cache benutzen kann, herangezogen werden [DFJ+96].

3.1 Replikationsfreie Sichten am Beispiel von DBProxy

DBProxy [APPT03] ist ein Verfahren von IBM, das SQL-Anfragen abfängt und überprüft, ob diese auch vom FE erfüllt werden können. Es entlastet den DBA, da dieser nicht ständig die im Cache befindlichen Tabellen kontrollieren und an die Anfragelast anpassen muss.

DBProxy ist als JDBC-Treiber implementiert, der von den FEs geladen wird. Trifft ein HTTP-Client-Auftrag auf dem FE-Server ein, der die Datenbank benötigt, so gibt der Server diesen an die Anwendungskomponenten weiter. Diese wiederum leiten die SQL-Anfrage, die die vom HTTP-Client-Auftrag benötigten Daten liefert, an die Datenbank durch den JDBC-Treiber weiter. Der JDBC-Treiber fängt, wie oben gesagt, die Anfrage ab und prüft, ob diese lokal befriedigt werden kann oder ob sie an den BE-Server weitergeleitet werden muss. Der Programmierer einer Applikation

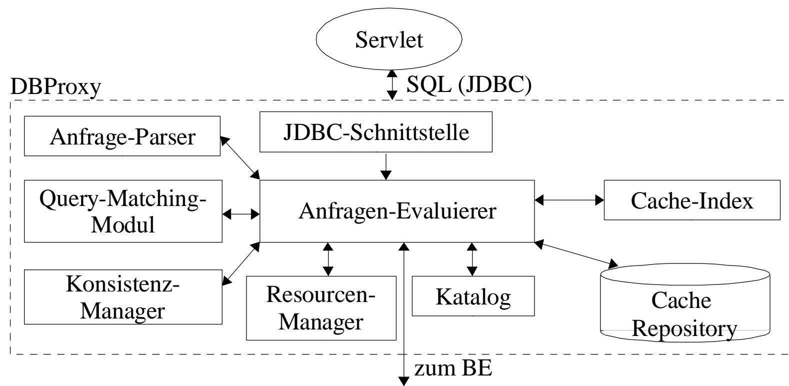

Abbildung 2: Die Architektur von DBProxy (nach [APPT03])

kann also wie gewohnt den BE-DBS ansprechen, da die Fragen von DBProxy abgefangen werden. Somit erfüllt DBProxy das Prinzip der *Transparenz.*

DBProxy ist selbst in mehrere Komponenten unterteilt (siehe Abbildung 2). Wird eine Anfrage von DBProxy abgefangen, so wird sie von der JDBC-Schnittstelle an das Herzstück von DBProxy weitergereicht – den Anfragen-Evaluierer. Diese enthält die Caching-Logik und entscheidet für jede Anfrage, ob der Zugriff ein Cache-Treffer ist oder nicht, indem er das Modul für das Matching von Anfragen (Anfragen-Matching-Modul) aufruft, das als Eingabe sowohl die Anfrageprädikate als auch die übrigen Klauseln der Anfrage erwartet. Der Anfragen-Evaluierer sorgt also in DBProxy für die Einhaltung der *Vollständigkeit.*

Ist eine Anfrage im Cache vorhanden, so kann sie lokal beantwortet werden, was in Abschnitt 3.1.2 noch genauer beschrieben wird. Ist sie noch nicht im Cache gespeichert, so entscheidet der Anfragen-Evaluierer, ob die Ergebnisse, die vom BE zurückgegeben werden, lokal gespeichert werden sollen oder nicht. Außerdem optimiert er Anfragen, die zum BE weitergereicht werden, um im Vorhinein Daten im Cache zu speichern und somit die Cache-Leistung zu verbessern.

Um die Cache-Inhalte und die Konfigurationsparameter dynamisch anpassen zu können, sammelt der Resource-Manager Statistiken über die Trefferraten und Antwortzeiten. Die *Konsistenz* des Caches wird vom Konsistenz-Manager überwacht. Ein Ablaufdiagramm für einen Cache Hit und Cache Miss kann Abbildung 3 entnommen werden. Es ist zu beachten, dass nicht immer Daten in den Cache gespeichert werden, wenn die Daten noch nicht im Cache vorhanden waren.

Im Folgenden werden einige der Komponenten genauer beschrieben. Zuerst wird das Cache Repository, in dem Anfrageergebnisse lokal gespeichert werden können, vorgestellt. Danach wird das Anfragen-Matching-Modul erläutert, das überprüft, ob eine Anfrage lokal ausgewertet kann, der Konsistenz-Manager, der die *Konsistenz* der Tupel im Cache gewährleisten soll und abschließend

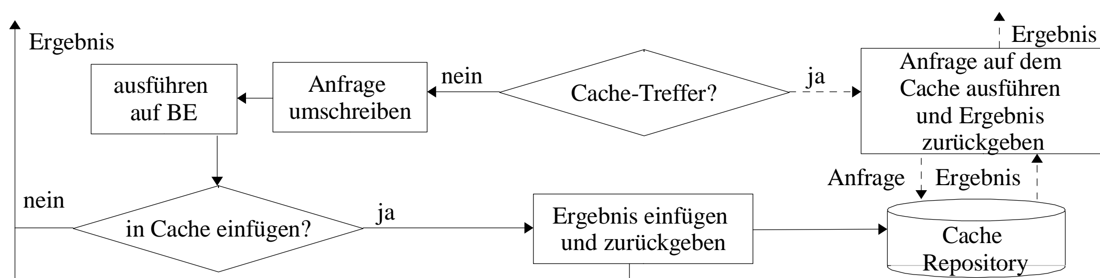

Abbildung 3: Verlauf eine Cache-Treffers und eines Cache Miss. Die gestrichelten Pfeile stehen für einen Cache-Treffer. (nach [APPT03])

Anfrage 1 (A1):
SELECT Name, Gehalt
FROM Personal
WHERE Gehalt >100k
AND Abt <5

Anfrage 2 (A2):
SELECT Gehalt
FROM Personal
WHERE Abt BETWEEN 4 AND 5

Personal auf dem BE

Id	Name	Gehalt	Abt
1	Hans Meiser	100k	4
2	Klaus Wald	200k	4
3	Peter Baum	100k	5
4	Manfred Vogel	150k	6
5	Oskar Grün	200k	3

$\xrightarrow{A1,A2}$

Personal auf dem FE

Id	Name	Gehalt	Durch
2	Klaus Wald	200k	**A1,A2**
5	Oskar Grün	200k	**A1**
1	*null*	100k	**A2**
3	*null*	100k	**A2**

Abbildung 4: Anfragen in DBProxy

die Garbage Collection im Cache, die nicht mehr benötigte Daten aus dem Cache entfernt, um den Platzverbrauch möglichst gering zu halten.

3.1.1 Cache Repository

Ein Cache-Index enthält eine Liste von Anfragen, welche den Inhalt des FE-Caches beschreiben. Um außerdem Platz zu sparen, werden, wann immer dies möglich ist – z. B. wenn sich Anfragen auf die gleichen Tabellen beziehen –, die Daten in einer einzigen Tabelle gespeichert, so dass mehrfache Anfrageergebnisse sich denselben (physischen) Platz teilen (siehe Abbildung 4). Diese Tabellen entsprechen jenen auf dem BE-Server, enthalten jedoch nur eine Teilmenge der Daten.

Join-Anfragen über die gleiche Basistabelle und mit der gleichen Join-Bedingung werden ebenfalls in der gleichen lokalen Tabelle gespeichert. Durch diesen Entwurf wird nicht nur eine effiziente Nutzung der physischen Ressourcen gewährleistet, sondern es wird auch leichter, *Konsistenz* zu gewährleisten.

Die lokalen Tabellen, die auf die Anfragen hin erstellt werden, enthalten so viele Spalten wie in der Originalanfrage ausgewählt wurden. Zusätzlich wird, falls nicht vorhanden, eine Spalte – oder auch mehrere, falls der Primärschlüssel aus mehr als einem Attribut besteht – mit dem Primärschlüssel der abgefragten Tabelle hinzugefügt, um bei weiteren Anfragen überprüfen zu können, ob ein Tupel eventuell schon in der lokalen Tabelle vorhanden sind und deswegen nicht nochmals eingefügt werden muss. Dies garantiert die *Konsistenz* des Caches, denn es verhindert eine redundante Speicherung von Daten. Durch solche Redundanzen wäre es nämlich möglich, dass nur eine Version des gleichen Tupels verändert oder gelöscht wird, während die andere unverändert im Cache verbleibt. Die Typen der Spalten und die Metadaten, also Informationen, die zum Erstellen der lokalen Tabelle notwendig sind, werden vom BE abgefragt und im lokalen Katalog gespeichert.

Abbildung 4 will dies veranschaulichen. Sie zeigt die Tabelle Personal auf dem BE-Server samt ihrer Inhalte. Auf dem FE ist die Tabelle nicht vorhanden. Id ist der Primärschlüssel von Personal. Wird nun die Anfrage A1 und danach Anfrage A2 auf der Tabelle Personal (siehe Abbildung 4) ausgeführt, so werden zuerst die Anfragen automatisch neu formuliert, damit sie die Primärschlüssel enthalten. In diesem Fall wird zur SELECT-Anweisung also Id hinzugefügt. Durch A1 werden die Sätze mit der Id 2 und 5 ausgewählt. Sodann wird eine Tabelle mit den in der veränderten SELECT-Anweisung enthaltenen Spalten erschaffen und anschließend die von A1 ausgewählten Sätze eingefügt. Da die Tabelle initial leer ist, werden alle Insert-Operationen ohne Fehler ausgeführt. Anschließend liefert A2 die Sätze mit der Id 1, 2 und 3.

Bevor die Sätze in die Tabelle eingefügt werden, überprüft DBProxy, ob zur Tabelle neue Spalten hinzugefügt werden müssen, was in diesem Beispiel nicht der Fall ist. Sodann wird der erste Satz mit der Id 1 eingefügt. In der Spalte „Name“ wird ein *null*-Wert eingefügt, da diese Spalte von A2 nicht

zurückgegeben wird und somit keine Informationen über den Inhalt existieren. Das heißt, dass ein eigentlich falscher *null*-Wert in die Tabelle eingefügt wird. Falsch ist dieser *null*-Wert deswegen, weil das Originaltupel auf dem BE-Server eigentlich keinen *null*-Wert in der Spalte Name enthält. Dieser falsche *null*-Wert könnte nun bei weiteren Anfragen auf die lokale Tabelle falsche Werte liefern, falls im SELECT die Spalte „Name“ ausgewählt wurde. Der Containment Checker von DBProxy verhindert dies allerdings. Anhand des Primärschlüssels Id ist nämlich feststellbar, dass sich der zweite Satz der Anfrage 2 mit Id 2 bereits in der lokalen Tabelle befindet. Somit würde eine Insert-Operation von der DB verworfen werden, die erkennt, dass das Tupel bereits vorhanden ist. DBProxy erkennt jedoch, dass das Tupel schon im Cache vorhanden ist und führt daher statt der Insert- ein Update-Operation durch. Der letzte Satz kann ebenfalls ohne Änderung eingefügt werden, enthält allerdings ebenfalls wieder einen falschen *null*-Wert.

Join-Anfragen werden ähnlich gehandhabt. Die Ergebnisse werden ebenfalls in einer einzigen Tabelle abgespeichert, die durch die Liste der Tabellen, die in der Join-Anfrage benutzt wurden, und durch die Join-Bedingungen beschrieben wird. Der Name der lokalen Tabelle für diese Anfrage wird im Cache-Index abgelegt.

Um die Anzahl der Änderungen des Schemas einer lokalen Tabelle zu reduzieren, wird initial der Fluss der Anfragen betrachtet, bis eine ausreichende Anzahl an chronologisch aufeinander folgenden Anfragen verfügbar ist, anhand derer die lokalen Tabellen vordefinierbar sind. Hier ist ein Mittelweg zu wählen zwischen dem Overhead, der dadurch entsteht, dass die komplette BE-Tabelle kopiert und der Platz für die Spalten allokiert, aber nicht genutzt wird und dem Overhead durch Schemaänderungen.

3.1.2 Anfragen-Matching-Modul

Das Anfragen-Matching-Modul ist dafür zuständig, zu überprüfen, ob eine beim FE-Server eintreffende Anfrage lokal beantwortet werden kann, also ob es in der Vergangenheit bereits Anfragen gab, in deren vereinigten Ergebnismengen die Ergebnismenge der neuen Anfrage enthalten ist.

Wie weiter oben schon erwähnt, werden Anfragen von DBProxy abgefangen und weiter zerlegt, indem die WHERE-Klausel als boolescher Ausdruck gespeichert wird. Die Blätter bilden dabei einfache Prädikate der Form „Spalte OP Wert“ bzw. „Spalte OP Spalte“, die durch AND-, OR- oder NOT-Knoten verbunden werden können. Ist dies geschehen, so wird ein Cache-Index verwendet, um die Anfragen herauszufiltern, die bereits auf der/den selben Tabelle/n ausgeführt wurden und welche eine Obermenge der Spalten, die von der zerlegten Anfrage benötigt werden, zurückgeliefert haben. Dann wird der Anfragen-Matcher aufgerufen, um zu überprüfen, ob die Ergebnismenge der neuen Anfrage in der Vereinigung (UNION) der Ergebnismengen der oben heraus gefilterten Anfragen enthalten ist. Um dies zu verifizieren geht der Matching-Algorithmus wie folgt vor:

Es ist zu überprüfen, ob die Ergebnismenge der Anfrage Q_B in der Ergebnismenge der Anfrage Q_A enthalten ist. Dies ist der Fall, wenn für alle möglichen Werte der Einträge in der Datenbank, das WHERE-Prädikat (where-P) der ersten Anfrage das WHERE-Prädikat der zweiten Anfrage logisch impliziert, also wenn where-P(Q_B) $\Rightarrow$ where-P(Q_A) gilt. Dies ist äquivalent zu der Aussage, dass where-P(Q_B) AND (NOT where-P(Q_A)) nicht erfüllbar ist.

Um das Beispiel von oben (siehe Abbildung 4 auf Seite 9) mit neuen Prädikaten weiterzuverwenden, wobei Q_B (Gehalt $<$ 150k) und Q_A (Gehalt $>$ 200k) entspricht, ist z. B. (Gehalt $<$ 150k AND (NOT Gehalt $>$ 200k)) erfüllbar, da als Lösung z. B. 100k möglich wäre. Also ist Q_B nicht in Q_A enthalten. Um diesen Ansatz auf die Vereinigung von mehreren Anfragen zu erweitern, ist es lediglich nötig, die verschiedenen WHERE-Klauseln mit einem logischen OR zu verbinden. Das in Abschnitt 3 erwähnte Semantic Caching wählt einen ähnlichen Weg, indem es auch anhand der schon gestellten Anfragen überprüft, ob eine neue Anfrage beantwortet werden kann [DFJ+96].

Es ist empirisch belegbar [APPT03], dass die meisten Anwendungen nur auf Templates basierende Anfragen benutzen, deren Auswahlprädikate die gleiche Struktur besitzen und sich somit nur in wenigen numerischen Werten oder String Konstanten unterscheiden (siehe Abbildung 5). Aus diesem Grund benutzt DBProxy einen Algorithmus, der genau dies ausnutzt, denn der oben beschriebene Al-

Template T	Anfragen nach Template T	
SELECT Shops FROM Niederlassungen WHERE plz = ?	SELECT Shops FROM Niederlassungen WHERE plz = 66763	SELECT Shops FROM Niederlassungen WHERE plz = 55469

Abbildung 5: Beispiele verschiedener Anfragen, denen das Template T zugrunde liegt

gorithmus produziert einen nicht zu vernachlässigenden Overhead, falls der Cache eine große Anzahl an Anfragen enthält. Dieser auf den oben beschrieben Templates basierende Anfragen-Matching-Algorithmus sammelt ähnliche Anfragen im Cache und mischt ihre Prädikate zusammen, indem er eine besondere Datenstuktur nutzt, die die Entwickler MAP (merged aggregate predicates) nennen. Trifft nun eine neue Anfrage auf, so wird zuerst überprüft, ob es im MAP schon eine solche Anfrage gibt. Ist dies der Fall, so können direkt die gewählten Daten aus dem Cache geladen werden. Damit wird der Overhead umgangen, der durch den weiter oben beschriebenen Matching-Algorithmus entstehen würde.

3.1.3 Konsistenz-Manager

Da die Daten in den FE-Tabellen lediglich Kopien der Daten in den BE-Tabellen sind, können Update-, Delete- und Insert-Operationen einfach zu den Tabellen im FE propagiert werden. Zukünftige Anfragen, die den Cache betreffen, bekommen also von diesen Änderung die jeweils passenden Tupel zurückgeliefert. Um festzustellen, ob eine Änderung den Cache betrifft, melden sich die Cache-Tabellen vorher beim BE-Server für die entsprechenden UDIs an.

Anfragen, die nur lesen, werden, soweit wie dies möglich ist, aus dem Cache beantwortet. Update-Operationen werden hingegen direkt zum BE weitergeleitet, damit diese dort ausgeführt werden können. Die Konsistenz der Daten wird dabei nur für den Cache überprüft – so weit dies möglich ist. Der BE-Server ist lediglich für die periodische Weitergabe der Update-Operationen zuständig – eine Aufgabe, die an einen separaten Prozess oder eine separate Maschine weitergegeben werden kann. Zu diesem Zweck sammelt ein Data Propagator alle UDIs, die an den BE-Tabellen ausgeführt wurden und leitet sie an die Cache-Tabellen auf den FE-Servern weiter, die sich für diese Änderungen beim BE-Server angemeldet haben. Die Änderungen werden in ihrer Commit-Reihenfolge auf dem Cache ausgeführt. Da immer Pakete von UDIs gesammelt werden, anstatt diese direkt am Cache anzuwenden, ändern sich die Daten nur langsam. Dies ist ein von den Entwicklern beabsichtigtes Verhalten, da sie annehmen, dass für die meisten Web-Umgebungen sich langsam ändernde Daten typisch sind. Ein Vorteil dieser Vorgehensweise ist, dass sich der Overhead, der durch die Bestimmung der zu aktualisierenden gecachten Sichten entstehen würde, verringert.

Obwohl bei DBProxy die Schwierigkeit besteht, dass sowohl propagierte Nachrichten (UDIs) als auch Ergebnisse von Anfragen, die ein Cache Miss waren, die Tabellen ändern können, garantiert DBProxy, dass der Zustand des Caches dem Zustand der BE-Tabellen entspricht, der d Zeiteinheiten zurück liegt (Lag Consistency). Denn Daten aus Anfragen, die ein Cache Miss waren und somit direkt vom BE-Server in den Cache eingetragen würden, wären aktuell. Die anderen Daten im Cache wären hingegen veraltet, weil die Veränderungen, die am BE-Server in der Zwischenzeit erfolgten, noch nicht im Cache vorgenommen wurden. Die Entwickler verhindern dies durch einen speziellen Algorithmus [APPT03], auf den hier wegen des begrenzten Platzes nicht weiter eingegangen wird.

3.1.4 Garbage Collection im Cache

Da es notwendig ist, alle UDIs, die am BE-Server durchgeführt wurden, zu den FE-Servern weiterzuleiten, kann es Tupel im Cache geben, die nicht zu einer gecachten Anfragen gehören. Da es außerdem Daten geben kann, die nicht (oder sehr selten) benutzt werden, besitzt DBProxy einen Garbage-Collection-Prozess, der im Hintergrund den Cache von nicht benötigten Daten reinigt, um

so den Nutzen des Caches für den begrenzten Speicherplatz optimiert. Beim Entfernen von Tupeln erfüllt DBProxy folgende Anforderungen [APPT03]:

- Wenn ein Tupel aus dem Cache entfernt wird, werden Tupel, die zu der selben Anfrage wie das gelöschte Tupel gehören, nur entfernt, wenn keine andere im Cache verbleibende Anfrage diese Tupel referenziert.
- Tupel, die durch den Konsistenz-Manager eingefügt werden und die nicht zur Ergebnismenge einer gecachten Anfrage gehören können eventuell von Garbage Collector entfernt werden.

Die Komponente, die für die Entfernung von Daten aus dem Cache verantwortlich ist, besteht aus der Replacement Policy, die bestimmt, was zu entfernen ist, und dem Replacement Mechanism, der bestimmt, wie die Daten entfernt werden sollen, damit die oben beschriebenen Anforderungen erfüllt sind. Das Löschen von Tupeln aus dem Cache wird gestartet, wenn der Platzverbrauch der DB einen hohen Pegel (high watermark, HWM) erreicht. Es werden darauf hin so lange Daten gelöscht, bis der Platzverbrauch unter einen gewissen Pegel (low watermark, LWM) sinkt. Da das Entfernen der Daten ein Hintergrundprozess ist, läuft dieser gleichzeitig mit der Beantwortung von Anfragen, Update-Operationen, und so weiter.

Als Beispiel für das Entfernen von Tupeln wird im Folgenden das Group Replacement beschrieben, welches schon die Entwickler von DBProxy in ihrem Artikel verwendeten [APPT03], da es sehr gut die Arbeitsweise des Algorithmus verdeutlicht und darüber hinaus auch gut verständlich ist.

Zuerst markiert die Replacement Policy, welche Tupel potentiell aus der DB gelöscht werden können. Dies geschieht durch das „accessed“-Flag[1], welches anzeigt, ob ein Tupel in letzter Zeit benutzt wurde. In diesem Fall besitzt das Flag den Wert *true*. Falls es den Wert *false* besitzt, so wurde es nicht benutzt und kann gelöscht werden. Damit nur Tupel gelöscht werden, die ausschließlich von „Opfer“-Anfragen[2] referenziert werden, werden folgende Schritte ausgeführt:

1. Das „accessed“-Flag wird für alle gecachten Spalten auf *false* gesetzt.
2. Alle Anfragen, die keine Opfer sind, werden im Hintergrund ausgeführt und setzen für jedes ausgewählte Tupel das entsprechende „accessed“-Flag auf *true*.
3. Sobald alle Anfragen ausgeführt wurden, werden alle Tupel gelöscht, deren Flag immer noch den Wert *false* besitzt.

Es kann nun jedoch das Problem auftreten, dass Tupel gelöscht werden, die während der Garbage Collection vom Konsistenz-Manager eingefügt oder verändert wurden. Um dieses Problem zu beheben, wird das „accessed“-Flag dieser Tupel auf *true* gesetzt, wodurch diese in Schritt 3 nicht entfernt werden. Somit garantiert der Garbage Collector, dass kein zu einer im Cache verbleibenden Anfrage gehörendes Tupel entfernt wird. Ebenso werden mit hoher Wahrscheinlichkeit die Tupel aus dem Cache entfernt, die extrem selten benutzt werden. Diese werden deswegen mit hoher Wahrscheinlichkeit gelöscht – und nicht mit einer Wahrscheinlichkeit von 1 –, weil es denkbar ist, dass gerade jene nicht aus dem Cache entfernt werden, da ihr „accessed“-Flag durch ein Update, o. ä. auf *true* geändert wurde. Durch Erfüllung dieser Voraussetzungen, liefert DBProxy *korrekte* und *vollständige* Tupel, die der BE-Server vor d Zeiteinheiten geliefert hätte.

3.2 Cache Groups am Beispiel von DBCache

DBCache [ABK+03, HB04a] ist ein DB-Caching-Verfahren, das ebenso wie DBProxy von IBM-Mitarbeitern entwickelt wurde. Es wurde als Prototyp auf der Basis der DB2 von IBM entwickelt, stellt also eine Erweiterung der DB2 dar. Der DBA muss jeweils die entsprechenden Tabellen auf dem

[1] Das „accessed“-Flag wird als zusätzliche Spalte an der jeweiligen Cache-Tabelle realisiert.

[2] Das sind die Anfragen, die gelöscht werden sollen.

Tabelle auf BE		**ProdNr bereichsv.**		**ProdNr nicht bereichsv.**	
ProdNr	Guete	ProdNr	Guete	ProdNr	Guete
1	A	1	A	1	A
1	B	1	B	1	B
2	A	2	A	*3*	A
3	A				
3	B				
3	C				

Abbildung 6: Bereichsvollständigkeit durch Cache Keys. Die Spalte ProdNr ist Cache Key.

FE-Server erschaffen, bzw. die Constraints festlegen – während DBProxy selbst für die Erstellung der Tabellen im Cache sorgte.

Zuerst werden die grundlegende Erstellung von Cache-Tabellen beschrieben, danach die Cache Constraints (CCs), welche die *Vollständigkeit* der Anfragen auf die Cache-Tabellen (CTn) gewährleisten sollen und schließlich die Probleme, die sich durch die Definition solcher CCs ergeben können. Darauf folgt eine Betrachtung, wie Anfragen in DBCache behandelt werden um abschließend auf die Wartung der CTn einzugehen.

3.2.1 Cache-Tabellen

Bei DBCache gibt es zwei Arten von Cache-Tabellen, die beide durch den DBA zu Beginn deklariert werden müssen. Die deklarativen CTn sind besonders dann geeignet, wenn der DBA schon zu Beginn weiß, welche Inhalte die Tabelle haben soll. Der Inhalt der Tabellen wird also schon vorab definiert. Um solche deklarativen CTn zu unterstützen, wurde das Konzept der materialisierten Sichten ausgenutzt und entsprechend erweitert. Die dynamischen CTn hingegen werden dynamisch anhand der Arbeitslast gefüllt (on-demand loading).

Um zu identifizieren, welcher Tabelle auf dem BE-Server die Cache-Tabelle entspricht, besitzt die CT einen „Spitznamen". Somit kann, falls eine Anfrage nicht von der CT beantwortet werden kann, die Anfrage durch den Spitznamen an die BE-Tabelle weitergeleitet werden. Der Spitzname entspricht dem Namen der zugehörigen BE-Tabelle. Wenn also die CT auf die Tabelle Personal verweist, so ist der Spitzname für Personal entsprechend Personal. Bis auf den Spitznamen sind die CTn in ihrem Schema identisch mit den jeweiligen Tabellen auf dem BE-Server. Dadurch können Daten, die im Cache gespeichert werden sollen, einfacher eingelagert werden. Ebenfalls besitzen CTn Indexe, Sichten und so weiter.

3.2.2 Cache Constraints

Cache Constraints wurden von den Entwicklern eingeführt, um *Vollständigkeit* (siehe Abschnitt 1) zu gewährleisten. Es gibt zwei verschiedene Arten von CCs: Cache Keys (CKs) und Referential Cache Constraints (RCCs). Momentan muss der DBA die CCs noch selbst bestimmen – dies wollen die Entwickler allerdings in Zukunft automatisieren [ABK+03].

Cache Keys Cache Keys sind Spalten, die als Einstiegspunkt für Anfragen und der Befüllung der Cache-Tabelle dienen. Damit CKs diese Eigenschaften erfüllen, müssen sie bereichsvollständig sein. Bereichsvollständigkeit bedeutet, dass für jeden Wert einer bereichsvollständigen Spalte die CT alle Zeilen aus der BE-Tabelle besitzt, die den gleichen Wert in der entsprechenden Spalte besitzen (siehe Abbildung 6). Somit sind alle Spalten, deren Werte *unique* sind – und damit auch der Primärschlüssel – implizit bereichsvollständig. Doch wie kann Bereichsvollständigkeit gewährleistet werden? Um eine Spalte bereichsvollständig zu machen, werden alle benötigten Tupel vom BE in die CT geladen. Im Falle der nicht bereichsvollständigen Tabelle aus Abbildung 6 wären dies die Tupel

(3, B) und (3, C). Die Bereichsvollständigkeit garantiert damit die Vollständigkeit von Gleichheitsprädikaten der Form $CT_i.c_x$ = Wert im Cache, wobei CT_i eine CT ist und c_x eine Spalte dieser CT. Ebenso wie die CTn selbst werden auch die CKs explizit erstellt.

Die bereichsvollständigen Spalten dienen als Einstiegspunkte für Anfragen. Das heißt, dass Anfragen mit einem Gleichheitsprädikat auf dieser Spalte korrekt im Cache beantwortet werden können.

Es gibt Bestrebungen, die recht restriktive Anforderung, Bereichsvollständigkeit für die Spalten zu sichern, aufzulockern. Die Idee ist, dass wertvollständige Tupel einer Spalte – das sind die Tupel, für die alle Tupel mit dem gleichen Wert in der BE-Tabelle auch in der FE-Tabelle vorhanden sind[3] – genügen, um die *Vollständigkeit* zu gewährleisten. Dafür muss gespeichert werden, welche Tupel wertvollständig sind. Diese Tupel könnten dann als Einstiegspunkt für eine Anfrage dienen.

Mit der bisherigen Definition können lediglich Anfragen auf nur eine einzige Tabelle ausgewertet werden. In der Praxis kommen allerdings Joins recht häufig vor – sie sollten also auch erfasst werden. Aus genau dieser Problematik heraus wurden die Referential Cache Constraints entwickelt.

Referential Cache Constraints Referential Cache Constraints definieren eine neue Art von für Cache-Datenbanken spezifischen Beziehungen. Ein RCC kann zwischen beliebigen Spalten verschiedener Tabellen beliebig definiert werden – abhängig davon ob ein Join zwischen ihnen möglich ist oder nicht. Der RCC erstellt eine Art Vater-Sohn-Beziehung zwischen zwei Cache-Tabellen. Ein RCC zwischen einer Spalte c_n einer Cache-Tabelle CT_i (Vater) und der Spalte c_m einer anderen Cache-Tabelle CT_j (Sohn) bedeutet, dass für jeden Wert in der Spalte c_n der Cache-Tabelle CT_i, CT_j alle Zeilen mit diesem Wert in der Spalte c_m enthält.

Cache Groups Eine Menge von CTn, die durch RCCs miteinander in Verbindung stehen, heißt Cache Group (CG). Diese Tabellen werden – direkt oder transitiv – durch den CK einer einzigen CT gefüllt. Dabei heißen die Tabellen, die direkt durch einen oder mehrere CKs gefüllt werden, Wurzel. Die Tabellen, die durch die auf ihnen definierten RCCs transitiv gefüllt werden, heißen Mitglied. Es ist möglich, dass sich CGs überlappen, falls die überlappenden CGs eine oder mehrere gemeinsame Mitglieder besitzen. Ist gar eine Wurzeltabelle einer CG Mitglied einer anderen CG, so kann eine Cache Goup die andere vollständig überdecken. Die Definition von Cache Groups ist wichtig, um den Kontext einer Anwendung zu verstehen und um frühzeitig Probleme zu erkennen, die durch die ungeschickte Wahl von CCs auftreten können (siehe Probleme durch Cache Constraints). Aus diesem Verhalten folgt, dass CTn, die weder CKs besitzen, noch Mitglied einer CG sind, vom System nicht befüllt werden können. In diesem Fall muss der DBA die Tabellen manuell füllen.

Um einen Überblick über eine oder mehrere CGs zu bekommen, kann eine CG durch einen sog. Cache-Group-Graphen dargestellt werden. Die Knoten dieses Graphen sind die CTn, deren CKs wiederum als Kommentare dem Knoten angehängt werden. CKs, die nicht *unique* sind, werden unterstrichen dargestellt. Die Kanten in diesem Graphen repräsentieren die RCCs. Bidirektionale Kanten bedeuten, dass zwischen zwei Spalten zweier Tabellen RCCs in beide Richtungen definiert wurden. Ein Pfad zwischen einer Start- und einer Endtabelle besteht aus den Kanten, die von der Starttabelle bis zur Endtabelle führen. Ein Pfad, bei dem Start- und Endtabelle übereinstimmen, wird Zyklus genannt. Somit ist jede bidirektionale Kante ein Zyklus.

Zyklen lassen sich wiederum in heterogene und homogene Zyklen unterteilen. Heterogene Zyklen sind Zyklen, bei dem zwei oder mehr Spalten in einer Tabelle zum Durchlauf des Zyklus benutzt werden (siehe Abbildung 7). Bei homogenen Zyklen wird nur jeweils eine Spalte im Durchlauf des Zyklus benutzt.

Probleme durch Cache Constraints Werden die CCs ungeschickt gewählt, so besteht die Gefahr, dass durch die RCCs zwischen den Tabellen eine erhebliche Anzahl an unerwünschten Daten in die CTn geladen werden. Dies kann im schlimmsten Fall dazu führen, dass die gesamten Inhalte der BE-Tabellen in ihre entsprechenden CTn gefüllt werden. Dies verursacht möglicherweise Platzprobleme

[3] Sind also alle Werte einer Spalte wertvollständig, so ist die Spalte bereichsvollständig.

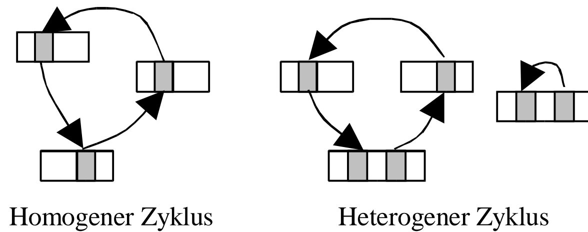

Abbildung 7: Homogene und heterogene Zyklen (nach [ABK+03])

auf den FE-Servern als auch eine hohe Aktualisierungslast. CCs, die ein solches Verhalten nicht verursachen, heißen sicher.

Es gibt zwei Regeln, die bei der Erstellung von CCs einzuhalten sind, um die oben beschriebenen unsicheren CCs zu verhindern:

1. Eine CG darf keine heterogenen Zyklen enthalten.
2. Eine CT darf nicht mehr als eine bereichsvollständige Spalte enthalten, deren Werte nicht *unique* sind.

Ein Beispiel zur ersten Regel findet sich in Abbildung 8. Wird in diesem Beispiel der Manager mit der Id 9 in den Cache geladen, so muss aufgrund des RCCs der Eintrag in der Tabelle Personal ebenfalls vorhanden sein. Da aber Manager 9 wiederum einen Manager mit Id 10 hat, wird dieser aufgrund des RCC von Personal auf Manager in der Tabelle Manager eingetragen, und so weiter. Dies kann im schlimmsten Fall so lange gehen, bis alle Datensätzen im Cache vorhanden sind – daher muss verhindert werden, dass ein heterogener Zyklus vorkommt.

Regel 2 lässt sich folgendermaßen zeigen: Wie in Abbildung 9 zu sehen, soll von der Originaltabelle Personal auf dem BE-Server der Datensatz mit der Id 3 in den Cache geladen werden. Damit PId in Personal auf dem FE bereichsvollständig bleibt, muss auch Id 4 nachgeladen werden (wegen Abt 2). Der Satz mit Id 5 muss in den Cache, damit die Spalte Gehalt bereichsvollständig bleibt, und so weiter. Es wird also in diesem Beispiel die gesamte Tabelle auf dem BE in den Cache geladen. In der Realität muss dies nicht unbedingt der Fall sein, aber dennoch verursacht dieses Verhalten eine mitunter große Menge an zu ladenden Daten und muss daher verhindet werden.

Manager

MId	...
6	
7	

Personal

PId	ManId	...
1	6	
2	6	
4	7	

Id 9 cachen →

Manager

MId	...
6	
7	
9	
10	
11	
⋮	

Personal

PId	ManId	...
1	6	
2	6	
4	7	
9	10	
10	11	
11	12	
⋮	⋮	

Abbildung 8: Beispiel zu Regel 1: Verhalten heterogener Zyklen (MId auf PId, ManId auf MId). In Manager soll ein neuer Satz mit Id 9 gecacht werden. Dieser hat den Manager mit Id 10 als Vorgesetzten, dieser Id 11, und so weiter.

Personal auf FE vor Caching

Id	Name	Abt	Gehalt
1	Herr A	1	100k
2	Frau B	1	200k

Personal auf BE
Personal auf FE nach Caching

Id	Name	Abt	Gehalt
1	Herr A	1	100k
2	Frau B	1	200k
3	Frau C	2	150k
4	Herr D	2	300k
5	Frau E	4	150k
6	Herr F	4	350k
7	Frau G	4	250k

Abbildung 9: Beispiel zu Regel 2 : mehr als eine bereichsvollständige Spalte (Abt und Gehalt)

3.2.3 Anfragen in DBCache

Wie oben angedeutet, sollen Anfragen entweder zum BE-Server oder zum FE-Server geleitet werden. Da es, wie schon in Abschnitt 3.2.1 beschrieben, zwei unterschiedliche Arten von CTn gibt, ist in DBCache für die Verarbeitung von Anfragen eine Unterscheidung zwischen beiden CTn nötig.

Anfragen auf deklarative Cache-Tabellen Wie schon in Abschnitt 3.2.1 erschöpfend dargelegt, werden deklarative CTn als eine Art materialisierte Sichten realisiert. Die Entwickler benutzen daher zum Beantworten der Anfragen auf deklarative Cache-Tabellen die existierenden Mechanismen zum Matching materialisierter Sichten (vergleiche auf Abschnitt 2.1).

Anfragen auf dynamische Cache-Tabellen Das Problem bei der Auswertung von Anfragen ist, dass aufgrund der dynamischen Natur der Cache-Tabellen zur Laufzeit entschieden werden muss, ob die CT zur Beantwortung der Anfrage verwendet werden kann.

Während der Kompilierung einer Anfrage werden zwei Pläne für jede Anfrage erschaffen:

- Ein lokaler Plan, der alle möglichen CTn und auch die Spitznamen anderer Tabellen enthält, die zur Beantwortung der Anfrage dienen können.
- Ein entfernter Plan, der nur aus Spitznamen besteht.

Beide Pläne werden durch einen Schalter („Switch Operator“) verbunden, der eine sog. Probe-Anfrage als Unteranfrage besitzt, die zur Laufzeit entscheidet, welcher der zwei Pläne benutzt werden soll. Diese Art von Plan nennen die Entwickler Janus-Plan, nach dem zweigesichtigen Gott der Römer.

Trifft nun eine Anfrage ein, so wird zuerst die Probe-Anfrage ausgeführt. Diese ist eine skalare Unteranfrage, deren Ergebnisse mit einem CHECK-Operator überprüft werden. Die Bereichsvollständigkeit garantiert nämlich, dass eine solche skalare Unteranfrage ausreichend ist, um die *Vollständigkeit* zu garantieren [ABK+03]. Findet die Probe-Anfrage heraus, dass der lokale Plan geeignet ist, die Anfrage zu beantworten – dies entspricht einem Cache-Treffer – so wird der lokale Plan ausgeführt. Der Janus-Plan sichert somit die *Vollständigkeit* der Daten im Cache für eine Anfrage.

Wie oben schon angedeutet beschränkt sich der Janus-Plan nicht nur darauf – wie dies bei DBProxy z. B. der Fall ist –, Anfragen komplett aus dem Cache zu beantworten. Vielmehr kann ein Teil der Anfrage aus dem Cache, der andere Teil aus dem BE-DBS beantwortet werden.

3.2.4 Wartung von Cache-Tabellen

In diesem Abschnitt wird gezeigt, wie die Cache-Tabellen in DBCache initial gefüllt und gewartet werden. Auch bei der Wartung der Cache-Tabellen ist zwischen deklarativen und dynamischen Cache-Tabellen zu unterscheiden.

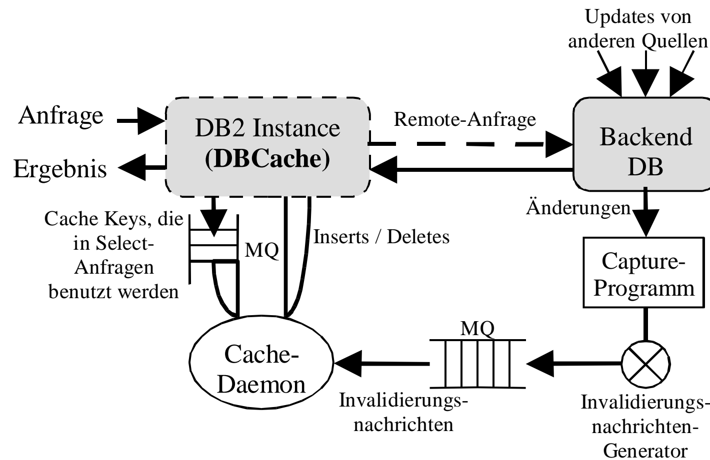

Abbildung 10: Der Cache-Daemon (nach [ABK+03])

Wartung von deklarativen Cache-Tabellen Die deklarativen CTn werden zu Beginn durch das DPropR-Programm gefüllt. Dieses IBM-Programm ist ein Programm zur Datenreplikation für relationale Daten. Es besteht aus zwei voneinander unabhängigen Programmen – einem, das Änderungen an den Daten feststellt und eines, das diese Änderungen ausführt.

Wartung von dynamischen Cache-Tabellen Wählt der Janus-Plan für eine Anfrage den entfernten Plan, so entspricht dies einem Cache Miss. Die CK-Werte, für die kein Wert in der CT existierte, werden extrahiert und dazu benutzt, den Cache zu aktualisieren. Dies geschieht jedoch nicht sofort, da sonst eine große Menge an Daten in den Cache geladen werden könnte, was sich negativ auf die Geschwindigkeit auswirkt. Aus diesem Grund werden die CK-Werte zusammen mit der Information, zu welcher Tabelle die CKs gehören, als nicht-persistente Nachrichten in eine Nachrichten-Queue eingereiht. Ein Cache-Daemon (siehe Abbildung 10), der als Prozess mit niedriger Priorität im Hintergrund läuft, leert die Queue. Zu diesem Zweck prüft er die CKs und kreiert entsprechende Update-Befehle, die er asynchron an die entsprechenden CTn weiterleitet. Die Idee, die hinter diesem Algorithmus steckt, ist, maximal einen Insert-Befehl pro Tabelle in einer Cache-Gruppe vorzubereiten um diese Befehle dann in einer Transaktion auszuführen. Die Befehle in dieser Transaktion werden in der Reihenfolge Cache-Vater – Cache-Kind ausgeführt. Die Vorbereitung der Befehle geschieht wie folgt:

1. Für jeden Wert eines empfangenen CK wird eine Menge von sich qualifizierenden Zeilen („Qualifying Rows“) bestimmt, die, unter Betrachtung aller CKs, in die Tabelle CT_0 eingefügt werden müssen. Dabei ist zu beachten, dass die Bereichsvollständigkeit für jeden CK gewahrt bleibt.

2. Beginnend bei CT_0 wird, basierend auf den sich qualifizierenden Zeilen von CT_i und den CKs, die auf CT_{i+1} definiert sind, für jeden RCC $CT_i \rightarrow CT_{i+1}$ eine Menge von sich qualifizierenden Zeilen bestimmt. Die Menge der sich qualifizierenden Zeilen für CT_{i+1} ist die Menge der Cache-Kind Zeilen entsprechend der Menge an sich qualifizierenden Zeilen vom CT_i plus alle weiteren Zeilen, die benötigt werden, um die Bereichsvollständigkeit entsprechend den CKs on CT_{i+1} zu wahren.

3. Wiederhole Schritt 2 rekursiv für alle ausgehenden Kanten von CT_{i+1}.

Es kann vorkommen, dass durch diesen Algorithmus Knoten, also Tabellen, mehrfach besucht werden, falls z. B. mehrere Kanten zu diesem Knoten führen. Entsprechend muss dann die Menge von sich qualifizierenden Zeilen erweitert werden.

Die sich qualifizierenden Zeilen einer CT werden durch (verschachtelte) Unteranfragen repräsentiert. Daher beinhaltet jeder Insert-Befehl für jede besuchte Tabelle eine Unteranfrage auf den Spitznamen, um alle sich qualifizierenden Zeilen zu finden, die noch nicht in der Tabelle sind [ABK+03].

Sollen die Inhalte in der CT aktualisiert oder gelöscht werden, weil die Sätze in der entsprechenden BE-Tabelle geändert wurden – was das weiter oben beschriebene DPropR-Programm registriert –, so generiert der BE-Server Invalidierungsnachrichten, die in eine Invalidierungs-Nachrichten-Queue eingefügt werden, die ebenfalls vom Cache-Daemon geleert wird (siehe Abbildung 10). Empfängt der Cache-Daemon eine solche Nachricht, so erstellt er Delete-Befehle entsprechend den CCs und leitet diese Befehle an die FE-DB weiter. Das Erstellen dieser Befehle kann unter Umständen recht komplex sein, wenn die aus den CCs resultierende Hierarchie sehr komplex ist. Sind die veralteten Zeilen aus dem Cache gelöscht, so können die aktuellen Spalten nachgeladen werden, wenn neue Anfragen verarbeitet werden. In der Zukunft planen die Entwickler, die Daten im Cache zu aktualisieren, statt sie zu invalidieren [ABK+03]. Der Cache-Daemon garantiert also die *Konsistenz* der dynamischen Tabellen.

4 Vergleich

In diesem Abschnitt werden die beiden Verfahren DBProxy, das in Abschnitt 3.1 vorgestellt wurde, und DBCache, das in Abschnitt 3.2 erläutert wurde, verglichen. Auf die statischen Verfahren wird im Folgenden nicht eingegangen, da sie keine neuen Konzepte einführen und sich – was wohl der wichtigste Grund ist – nicht dynamisch an die Anfragelast anpassen können.

Der auffallendste – und oberflächlichste – Unterschied zwischen DBProxy und DBCache ist sicherlich die unterschiedliche Art der Implementierung. Während DBCache an die IBM-Datenbank-Distribution DB2 gebunden ist, ist DBProxy von der jeweiligen dahinter liegenden DB unabhängig, da es als JDBC-Treiber implementiert wurde, der im Hintergrund für das erwünschte Caching-Verhalten sorgt.

Große Unterschiede bestehen bei beiden Verfahren sicherlich auch im Matching von Anfragen. DBProxy speichert die Anfragen im Cache und vergleicht neu eintreffende Anfragen mit diesen. DBCache hingegen überprüft mit einer Unteranfrage, ob der Wert der Spalte, die in der Where-Klausel der Anfrage verwendet wurde schon im Cache vorhanden ist. Die Bereichsvollständigkeit garantiert dann nämlich, dass alle Werte, die die Anfrage benötigt, sich im Cache befinden. Die *Konsistenz* des Caches muss bei DBProxy nicht weiter kontrolliert werden, bei DBCache dagegen müssen häufig weitere Werte in den Cache geladen werden, wodurch die in Abschnitt 3.2.2 beschriebenen Probleme auftreten können. Allerdings ist es in DBCache möglich, Teile der Anfrage aus dem Cache zu beantworten, während DBProxy entweder die Anfrage komplett aus dem Cache oder komplett aus dem BE-DBS beantwortet. Hier zahlt sich die Konsistent-Haltung des Caches in DBCache aus. Beide Verfahren fügen aber – gegebenenfalls zeitlich verzögert – Ergebnisse einer Anfrage, die aus dem BE-DBS beantwortet wurde, in den Cache ein.

Wie im obigen Absatz bereits erwähnt, gibt es bei DBProxy keine Konsistenzen beim Einfügen von Tupel zu wahren. Das einzige, was der Konsistenz-Manager hier erledigen muss, ist dafür zu sorgen, dass alle UDIs am BE auch am FE durchgeführt werden. Diese Art der Konsistent-Haltung erledigt in DBCache der Cache-Daemon. Des weiteren muss DBCache jedoch auch dafür Sorge tragen, dass falls neue Tupel in den Cache geladen werden, Tupel mit dem gleichen Wert ebenfalls aus dem BE in den Cache geladen werden, um Bereichsvollständigkeit zu garantieren.

Beide Verfahren gehen außerdem in der Wartung des Caches unterschiedliche Werte. So sammelt DBProxy, falls der Platzverbrauch der FE-DB einen zu hohen Wert annimmt, durch den Garbage Collector jene Tupel, die von keiner im Cache verbleibenden Anfrage referenziert werden und entfernt diese anschließend. Dieser Vorgang wird so lange fortgeführt, bis genügend Platz freigeworden ist. DBCache verfügt über keinen solchen Mechanismus, sodass denkbar ist, dass der Cache in schlimmsten Fall alle Tupel, die auch im BE-DBS enthalten sind, enthält. Allerdings wirkt das kaskadierte Löschen der Tupel durch den Cache-Daemon dem ein wenig entgegen.

Ein weiterer Unterschied zwischen beiden Verfahren besteht in der Behandlung von Templates (siehe Abschnitt 3.1.2). DBProxy nutzt zur Behandlung von Templates einen Algorithmus, der ähnliche Anfragen im Cache sammelt und deren Prädikate im MAP zusammen mischt [APPT03]. Wird eine neue Template-Anfrage an den Cache gestellt, so erkennt DBProxy dies durch MAP und beantwortet sofort. Damit wird die Überprüfung, ob das Ergebnis einer Anfrage bereits im Cache ist, umgangen. DBCache nutzt dieses Prinzip nicht aus. Es ist stets notwendig, den Janus-Plan zu erstellen und dann mittels Unteranfrage zu überprüfen, ob einige der Werte bereits im Cache sind.

Aus den oben beschriebenen Unterschieden lassen sich verschiedene zugrunde liegende Konzepte für beiden Verfahren erkennen.

DBProxy ist sehr darauf ausgelegt, möglichst wenig Daten im Cache zu speichern. Damit ist DBProxy für FE-Server, die nur wenig Speicher zur Verfügung haben, besser geeignet als DBCache. Dem entgegen steht bei DBCache jedoch, dass Anfragen nicht ausschließlich aus dem Cache beantwortet werden müssen, sondern ein Teil weiter an den BE geleitet werden können, während DBProxy nach dem Alles-oder-nichts-Prinzip vorgeht. Es ist daher denkbar, dass sich bei sehr vielen Anfragen, die sich in den gewählten Spalten unterscheiden, der Vorteil von DBProxy auf ein Minimum reduziert, da stets die Anfragen an den BE weitergeleitet werden. Die dabei ausgewählten Daten werden zwar in den Cache eingefügt und nach einiger Zeit müsste der Template-Algorithmus dafür sorgen, dass die Anfragen aus dem Cache beantwortet werden können, aber bei ungünstig gewählter HWM (siehe Abschnitt 3.1.4) könnten diese Werte gelöscht werden, bevor sie sinnvoll genutzt werden können.

DBCache verbraucht zwar rein von der Konzeption des Verfahrens her mehr Speicherplatz, ist allerdings durch die Konsistent-Haltung des Caches sauberer, da mit der Anzahl der Einträge auch die Anzahl der Möglichkeiten steigt, Anfragen im Cache auszuwerten, weil die Anzahl der Tupel mit unterschiedlichen Werten in den Einstiegsspalten aller Tabellen steigt. Zudem könnte der Overhead durch das Nachladen von Werten weiter reduziert werden, wenn sich die Forderung durchsetzt, Wertvollständigkeit statt Bereichsvollständigkeit sicherzustellen.

5 Fazit

Abschließend lässt sich feststellen, dass alle Verfahren – mit Ausnahme des Full-Table-Cachings – die in der Einleitung geforderten Eigenschaften wie *Konsistenz*, *Vollständigkeit* und *Transparenz*, erfüllen. Somit sind all diese Verfahren zum DB-Caching geeignet. Dennoch sind für ein DB-Caching, das möglichst effizient sein will, die statischen Realisierungen denkbar ungeeignet, da sie keinen Mechanismus besitzen, um sich an die Anfragelast anpassen zu können. Weitere Algorithmen zum DB-Caching könnten jedoch auf ihnen aufbauen, wie dies auch bei DBCache der Fall ist, welches das Prinzip der MSen für sich nutzt.

In Zukunft ist eine Weiterentwicklung der hier vorgestellten dynamischen Verfahren zu erwarten, was die in Abschnitt 4 beschriebenen Unterschiede mehr oder minder relativieren wird. Zu erwarten wäre z. B. eine Erweiterung von DBCache um einen Garbage Collector oder ähnliches. Es besteht also noch Forschungsbedarf bei beiden Verfahren, da auch die Herausforderungen an ein DB-Caching durch eine vermehrte Nutzung von Online-Warenhäusern stetig steigen. Damit bleibt eine Betrachtung der Entwicklung dieser Verfahren interessant.

Literaturverzeichnis

[ABK+03] Altinel, Mehmet ; Bornhövd, Christof ; Krishnamurthy, Sailesh ; Mohan, C. ; Pirahesh, Hamid ; Reinwald, Berthold: Cache Tables: Paving the Way for an Adaptive Database Cache, Proc. 29th Int. Conf. on Very Large Data Bases, Berlin, 2003, S. 718–729

[Ada04] Adam, Sebastian: Web-Caching, Kaiserslautern, Technische Universität, Fachbereich Informatik, Seminararbeit, 2004

[APPT03] Amiri, Khalil ; Park, Sanghyun ; Padmanabhan, Sriram ; Tewari, Renu: DBProxy: A dynamic data cache for Web applications, Proc. Int. Conf. on Data Engineering, Bangalore, India, 2003, S. 821–831

[BDD+98] Bello, Randall G. ; Dias, Karl ; Downing, Alan ; Feenan, James ; Finnerty, Jim ; Norcott, William D. ; Sun, Harry ; Witkowski, Andrew ; Ziauddin, Mohamed: Materialized Views In Oracle, Proc. 24th Int. Conf. on Very Large Data Bases, New York, 1998, S. 331–341

[DFJ+96] Dar, Shaul ; Franklin, Michael J. ; Jonsson, Björn T. ; Srivastava, Divesh ; Tan, Michael: Semantic Data Caching And Replacement, Proc. 22th Int. Conf. on Very Large Data Bases, Bombay, 1996, S. 330–341

[GL01] Goldstein, Jonathan ; Larson, Per-Ake: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution, Proc. Int. Conf. on Management of Data 2001, Santa Barbara, CA, 2001, S. 659–664

[HB04a] Härder, Theo ; Bühmann, Andreas: Database Caching: Towards a Cost Model for Populating Cache Groups, Proc. 8th East Europ. Conf. ADBIS, Budapest, Hungary, 2004

[HB04b] Härder, Theo ; Bühmann, Andreas: Datenbank-Caching – Eine systematische Analyse möglicher Verfahren. In: *Informatik – Forschung und Entwicklung (noch nicht erschienen)* (2004)