Datenbank-Caching –
Eine systematische Analyse möglicher Verfahren

Theo Härder, Andreas Bühmann
Technische Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
e-mail: haerder/buehmann@informatik.uni-kl.de

Zusammenfassung. Caching ist ein bewährtes Mittel, um die Skalier- und Verfügbarkeit von Systemen zu steigern sowie die Latenzzeit für Benutzeranforderungen zu verkürzen. Im Gegensatz zum Web-Caching, bei dem einzelne Web-Objekte irgendwo längs ihres Aufrufpfades in der Proxy-Kette vorgehalten werden, setzt Datenbank-Caching „ausgewachsene" Datenbanksysteme als Caches ein, um dort Satzmengen entfernter Datenbanken möglichst adaptiv verwalten und Anfragen darauf auswerten zu können. Verfahren dazu reichen von separat verwalteten materialisierten Sichten über überlappende, aber replikationsfrei gespeicherte Sichten bis hin zu Cache-Groups, in denen parametrisierte Cache-Constraints den Cache-Inhalt spezifizieren. Wir untersuchen anschaulich die verschiedenen Ansätze und ermitteln daraus eine Klassifikation, die den Lösungsraum zu enthüllen hilft. In den Mittelpunkt stellt sich das Konzept der Prädikatsvollständigkeit: Ein Datenbank-Cache verwaltet vollständige Extensionen von Prädikaten, was ihm ermöglicht, Schlüsse über beantwortbare Anfragen zu ziehen.

Schlüsselwörter. Datenbanksysteme, Datenbank-Caching, Anfrageverarbeitung, Materialisierte Sichten, Prädikatsvollständigkeit

Abstract. Caching is a proven remedy to enhance scalability and availability of software systems as well as to reduce latency of user requests. In contrast to Web caching where single Web objects are kept ready somewhere along their invocation path in the proxy chain, database caching uses full-fledged database management systems as caches to adaptively maintain sets of records from a remote database and to evaluate queries on them. This proceeding can be achieved by a spectrum of techniques ranging from separated materialized views via overlapping, but „replication-free" stored views up to cache groups where parameterized cache constraints specify the contents of the cache. We explore different approaches in an illustrative way and, from the insights gained, we derive a classification scheme which helps to discover the solution space. The key concept for advanced techniques is predicate completeness: A database cache maintains complete extensions of predicates which enables the system to draw conclusions about locally answerable queries.

Keywords. Database systems, database caching, query processing, materialized views, predicate completeness

CR Subject Classification: C.2.4, D.4.4, H.2.4, H.2.8

1 Motivation

e-Applikationen für die verschiedensten Bereiche (e-*) entstehen und wachsen in Anzahl und Größe in dramatischer Weise. Deshalb stehen bei deren Systementwicklung Anforderungen wie Skalierbarkeit, Leistungsverhalten – insbesondere Minimierung der Latenzzeit für den Endbenutzer (WWW-Browser, Web-Client) – und Verfügbarkeit an vorderster Stelle. In der Vergangenheit waren bei diesen Applikationen die durchs Web zu transportierenden Objekte (Web-Objekte wie HTML-Seiten, XML-Fragmente, Bilder, Videos usw.) weitgehend statisch, so dass, um die obigen Anforderungen zu erfüllen, ein Caching dieser Web-Objekte irgendwo längs ihres Aufrufweges hilfreich war: Cache des Web-Clients, (forward and reverse) Proxy-Caches, CDNs (Content-Delivery Nodes) oder spezielle Objekt-Caches der Anwendungen. CDNs wie z. B. Akamai's EdgeSuite [1] und IBMs Websphere Edge Server [10] erlauben das Verschieben von Anwendungskomponenten (Servlets, Java Server Pages, Enterprise Beans und Page Assembly) in diese so genannten Edge Server, um die in einem Cache gefundenen (statischen) Web-Objekte möglichst nahe beim Web-Client zusammenbauen und an den Benutzer ausliefern zu können.

Dieses so genannte *Web-Caching* [18] bietet für die in e-Applikationen auftretenden Lastcharakteristika bei statischen Web-Objekten die allgemeinen Vorteile von Caching:

- Entlastung der Kommunikationsverbindungen und Reduktion der benötigten Übertragungsbandbreiten
- Verkürzung der Antwortzeit für die Aufrufe durch Web-Benutzer
- Entlastung des (eigentlich aufgerufenen) Web-Servers.

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 [16].

In jüngster Zeit präsentieren e-Applikationen jedoch immer mehr dynamisch generierte Inhalte, da Schritt für Schritt eine immer stärkere Benutzerorientierung angestrebt und auch möglich geworden ist (personalisierte Web-Seiten, zielgerichtete Werbung, interaktives e-Commerce usw.). Dynamische Inhalte können jedoch nicht sinnvoll und kosteneffektiv in Web-Caches genutzt werden, die sich irgendwo im Web und weit entfernt von den Applikationsservern befinden, welche erst mit Hilfe von Zugriffen auf einen DB-Server (Backend-(BE-)DB-Server oder kurz BE-DBS) diese Inhalte zeitnah generieren müssen. Die verstärkte Bereitstellung von aktuellem, d. h. dynamischem Inhalt führt deshalb einerseits dazu, dass transaktionale Web-Applikationen auf der Basis unternehmensweiter Anwendungsinfrastrukturen (weltweit verteilte Middleware) realisiert werden, die wiederum eine (heterogene) Vielfalt an Technologien wie Mechanismen zur Lastbalancierung im Netz, Web-Server, Applikationsserver, TP-Monitore und Datenbanken voraussetzen. Andererseits erfordert die Dynamik der Anforderungen, dass die Applikationsserver in entfernte Datenzentren in der Nähe der Endbenutzer auswandern (auch Web-Hosting-Dienste genannt), um ihre Dienstleistungen „vor Ort“ anbieten zu können. Wegen der häufigen Zugriffe auf den dann entfernten (zentralen) BE-DB-Server führt diese Lösung allein zu einer Verschlechterung aller wesentlichen Verarbeitungs- und Betriebsaspekte (Skalierbarkeit, Antwortzeit, Verfügbarkeit), da eine Benutzeranforderung an den zwar nahen Applikationsserver oft eine Vielzahl von entfernten DB-Aufrufen nach sich zieht.

In solchen Situationen verspricht das so genannte *Datenbank-Caching* effektive Abhilfe. Die am einfachsten zu implementierende Lösung wäre es, für transaktionale Web-Applikationen den ganzen Inhalt von ausgewählten BE-Tabellen (statisch) in den verschiedenen Frontend-(FE-)DB-Servern (kurz FE-DBS) zu replizieren [15], was auch als Full-Table Caching bezeichnet wird. Bei großen Tabellen kann sich diese Vorgehensweise schon bei moderater Änderungsdynamik wegen der erhöhten Replikations- und Wartungskosten als nicht machbar herausstellen. Deshalb ist dynamisches Datenbank-Caching erfolgsversprechender, welches die demnächst benötigten Daten „in der Nähe“ des Applikationsservers – und zwar transparent für das Anwendungsprogramm – bereitstellt, um Anfragen lokal ausführen zu können.

Neue Herausforderungen ergeben sich hier wegen des typischen SQL-Einsatzes durch die daraus resultierende deklarative und mengenorientierte DB-Verarbeitung. Das bedeutet, dass stets gewährleistet sein muss, dass (Teil-)Anfragen im DB-Cache abgewickelt werden können, d. h., dass die Satzmengen der BE-DB, welche die zugehörigen SQL-Prädikate erfüllen, sich vollständig im DB-Cache befinden. Im Folgenden bezeichnen wir eine solche Satzmenge als *Extension* des entsprechenden Prädikats. Weiterhin müssen diese Extensionen konsistent zur BE-DB gehalten werden. Aktualisierungsoperationen treffen von mehreren FE-DBS im BE-DBS ein und werden dort ausgeführt. Da sich gleiche oder überlappende Extensionen in mehreren FE-DBS befinden können, müssen Änderungen an alle FE-DBS weitergeleitet werden, die eine betroffene Extension speichern.

Das Szenario für eine solche Verarbeitung ist in Abb. 1 illustriert. Algorithmen zur Web-Server-Auswahl gestatten den Web-Clients, aus der Menge der replizierten Server einen geeigneten zu bestimmen, der „nahe“ bei ihnen ist und dadurch die Antwortzeit für die Web-Dienste minimiert. Damit geht die Erwartung einher, dass der Web-Server Applikationsserver nutzen kann, die Dienste und Daten mit geographischem Kontext effizient bereitstellen können. In Abb. 1 wurde durch die Beispielprädikate an den DB-Caches angedeutet, dass dort oft Daten mit lokalem Bezug zum Applikationsserver und den dort erwarteten Anfragen vorgehalten werden. Wegen der deklarativen API (Application Programming Interface) der transaktionalen Web-Applikationen ist es unabdingbar, dass bei jeder Anfrageauswertung die *Vollständigkeitsbedingung* eingehalten wird, d. h., dass für das Anfrage(teil)prädikat die zugehörige Extension in der FE-DB ist. Zur flexibleren und kosteneffekti-

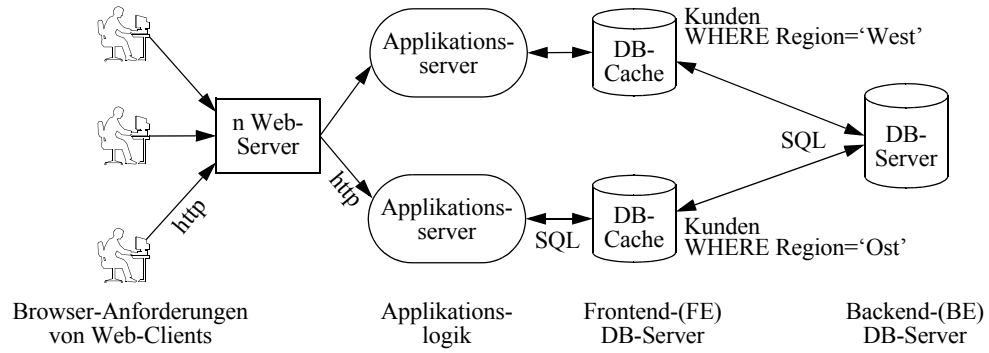


Abb. 1: Datenbank-Caching für Web-Anwendungen

veren Nutzung des Verfahrens wollen wir nicht fordern, dass immer das vollständige Anfrageprädikat im FE-DBS ausgewertet werden muss. (Teil-)Prädikate, deren Auswertung lokal nicht möglich ist, sind zur BE-DB zu schicken. Ihre Ergebnismenge wird dann mit der lokal ermittelten Satzmenge im FE-DBS zum Anfrageergebnis zusammengebaut, bevor dieses an den Benutzer übermittelt wird. Der begrifflichen Einfachheit und Einheitlichkeit halber verzichten wir im Folgenden auf die Differenzierung zwischen (Teil-)Anfragen und (Teil-)Prädikaten bei der Beschreibung der Verarbeitung im DB-Cache.

Caching wird in vielen Situationen als Allheilmittel für die Entlastung von Kommunikationsnetzen, die Minimierung von Antwortzeiten sowie die Skalierbarkeit empfohlen. Die Nutzung von DB-Caching, entfernt vom BE-DB-Server – in Middleware-Komponenten der Unternehmensinfrastruktur, in Datenzentren oder Edge-Servern –, soll Verbesserungen hinsichtlich dieser Entwurfs- und Einsatzziele bringen. Als unabdingbare Eigenschaft sehen wir dabei die Auswertung von (komplexen) Anfragen im DB-Cache an. Für sinnvolle Lösungen unterstellen wir, dass die Änderungsdynamik in der BE-DB zu hoch ist, um Replikation von Tabellen oder Sichten kosten- und zeiteffektiv vornehmen zu können, aber andererseits die Häufigkeit der Änderungen moderat genug ist, um „entferntes“ Caching überhaupt zu gestatten.

In diesem Beitrag durchleuchten wir in Abschnitt 2 die Unterschiede zwischen DB-Pufferverwaltung und DB-Caching, bevor wir in Abschnitt 3 existierende Verfahren zum DB-Caching beschreiben und in ihren wesentlichen Eigenschaften und Merkmalen analysieren. Aus Platzgründen skizzieren wir die Anforderungen zur Konsistenzerhaltung der DB-Caches nur. Aus den dabei gewonnenen Einsichten leiten wir danach ein Klassifikationsschema ab, das uns Hinweise auf weitere mögliche Lösungen für das DB-Caching liefert, bevor wir in Abschnitt 5 unsere Ergebnisse zusammenfassen und mit einem Ausblick unsere Betrachtungen abschließen.

2 DB-Pufferverwaltung und DB-Caching

Beim DB-Caching sollen also DB-Daten insbesondere für Web-Anwendungen in mehreren FE-DBS „in der Nähe der und transparent für die Anwendungen“ verarbeitet werden, d. h., durch die räumliche Nähe soll die SQL-Verarbeitung beschleunigt werden, ohne dass die SQL-API in irgendeiner Weise modifiziert werden muss. Um die dabei entstehenden Probleme des Caching besser verstehen zu können, sollen zunächst die Unterschiede zwischen der DB-Pufferverwaltung (im BE-DBS und in den FE-DBS) und dem DB-Caching in den FE-DBS genauer herausgearbeitet werden.

2.1 Wirkungsweise der DB-Pufferverwaltung

Die DB-Pufferverwaltung hat innerhalb eines DBMS die enorm wichtige und leistungskritische Aufgabe, eine kosteneffektive Geschwindigkeitsanpassung der Externspeicherzugriffe an die Verarbeitung der DB-Daten im Hauptspeicher zu erzielen. Im Prinzip soll dadurch das inhärente Geschwindigkeitsgefälle zwischen verschiedenartigen Speichermedien innerhalb einer Speicherhierarchie verdeckt werden – hier zwischen dem flüchtigen, instruktionsadressierbaren elektronischen Speicher (RAM) und dem nichtflüchtigen, blockadressierbaren magnetischen Speicher (Magnetplatte) mit seiner hohen Latenzzeit, oft als „Zugriffslücke“ bezeichnet. In Abb. 2 sind für die DB-Pufferverwaltung die wesentlichen Komponenten zur Erklärung ihrer Wirkungsweise illustriert.

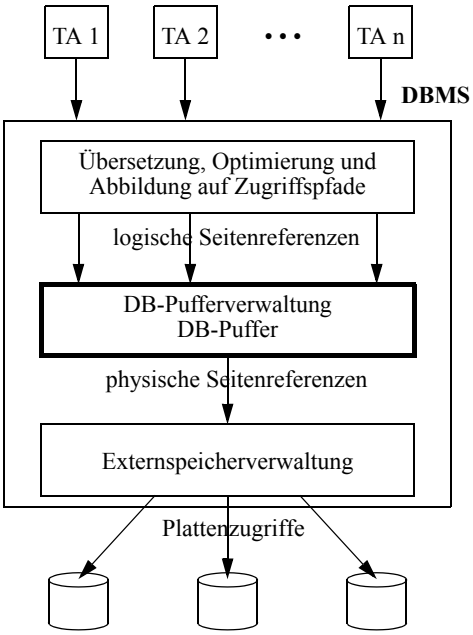


Abb. 2: Stellung der DB-Pufferverwaltung innerhalb einer DBMS-Architektur

Transaktionsprogramme (TA i) schicken an das DBMS deklarative und mengenorientierte SQL-Anweisungen, die durch die Übersetzungs- und Optimierungskomponente auf die im DBS vorhandenen Zugriffspfade abgebildet und als Zugriffsmodule abgewickelt werden. Ein Zugriffsmodul verkörpert im DBMS eine SQL-Anweisung; bei ihrer Ausführung fordert das Modul über logische Seitenreferenzen die benötigten DB-Daten bei der DB-Pufferverwaltung an. Wenn die referenzierte Seite sich bereits im DB-Puffer befindet, wird der lange Weg zum Externspeicher vermieden. Ein Seitenzugriff mit einer kurzen Analyse des Seiteninhalts dauert in diesem Fall ~ 100 ns. Muss eine Seite dagegen vom Externspeicher geholt werden, ergibt sich eine physische Seitenreferenz zum Bereitstellen der Seite, die einhergeht mit einer Seitenersetzung und oft die Auslagerung einer geänderten Seite impliziert, da im DB-Puffer erst Platz für die neue Seite geschaffen werden muss. Für eine solche Aktion sind in der Regel 20–30 ms anzusetzen, was einer Verteuerung um den Faktor $2 \cdot 10^5$ entspricht. Durch Prefetching kann man versuchen, physische Seitenreferenzen asynchron zur Ausführung der betreffenden SQL-Anweisung zu bearbeiten, um ihre Auswirkungen auf die Antwortzeit abzumildern. Bei On-demand-Anforderungen geht dagegen jeder physischen eine logische Seitenreferenz voraus. Jedoch führt (glücklicherweise) nicht jede logische zu einer physischen Seitenreferenz, da die DB-Pufferverwaltung die zeitliche und räumliche Lokalität der Seitenreferenzen ausnutzt und diese durch geeignete Seitenersetzungsalgorithmen verstärkt mit dem Idealziel, dass sich die in nächster Zukunft benötigten Seiten immer bereits im DB-Puffer befinden, d. h., ihr Optimierungsziel ist die Minimierung der physischen Seitenreferenzen [8].

In großen DBS übertrifft das Volumen der DB-Daten die DB-Pufferkapazität trotz ihres starken Wachstums[1] oft um den Faktor 1000. Wären die logischen Seitenreferenzen lokalitätsfrei (zufällig und gleichverteilt), so müsste man mit einer Trefferrate im DB-Puffer von 0,1 % rechnen. Tatsächlich jedoch zeichnet eine gute DB-Pufferverwaltung aus, dass sie – vor allem durch effektive Ausnutzung von Lokalität – in der Regel Trefferraten von > 95% gewährleistet.

Natürlich spielt für das gesamte DBS-Leistungsverhalten auch eine effektive Unterstützung der Datensuche durch zugeschnittene Zugriffspfade eine wesentliche Rolle. Betrachten wir eine einfache SQL-Anfrage mit Primärschlüssel-Suchbedingung auf einer Tabelle. Ist die Tabelle als Hash-Struktur abgelegt, wird der Optimierer daraus einen Zugriffsplan gestalten, der mit einer einzigen logischen Seitenreferenz auskommt. Liegt ein Index für den Primärschlüssel vor, realisiert als B*-Baum der Höhe 3, reichen zur Lokalisierung des Satzes 4 logische Seitenreferenzen aus. Ist dagegen kein brauchbarer Zugriffspfad zu finden, muss ein Tabellen-Scan angestoßen werden, der typischerweise eine große Anzahl von logischen Seitenreferenzen auslöst (oft 10^4–10^6), deren Auswirkung auf die Antwortzeit dann nur durch Prefetching abgemildert werden kann.

Im Prinzip erfordert die Leistungsoptimierung von DB-Anwendungen (physischer DB-Entwurf) neben Überlegungen zu Cluster-Bildung die Einführung von zur Anfragelast passenden Zugriffspfaden, wodurch bereits ein Großteil der Anwendungslokalität, d. h. der Lokalität der logischen Seitenreferenzen, festgelegt wird. Eine anwendungsspezifische Optimierung des DB-Puffers kann durch eine gezielte Wahl des Ersetzungsalgorithmus erfolgen, z. B. durch LRU-K, LRD oder CLOCK (vor allem bei sehr großer Rahmenanzahl) [8], wenn etwas über die Zugriffslokalität der Anwendung bekannt ist.

1. Auch hier gilt das Moore'sche Gesetz mit einem Wachstumsfaktor von 100 in jeweils 10 Jahren seit 1970. Setzt man die Pufferkapazität von DBMS 1970 (virtuell) mit 1 an (bei einer Rahmengröße von 2KB), so lag sie 1975 typischerweise bei 10, 1985 bei 1000 und 1995 bei 10^5. Im Jahr 2005 kann man (in der Spitze) mit Pufferkapazitäten von 10^7 (bei einer Rahmengröße von 8KB und mehr) rechnen.

Damit ist aber schon die einzige Entscheidung (neben der Wahl der Kapazität) angesprochen, die für den Betrieb des DB-Puffers zu fällen ist. Alle anderen Betriebs- und Verarbeitungsaspekte sind vorgegeben. Die einfache Pufferschnittstelle „stelle Seite x bereit“ bestimmt Granulat und Zeitpunkt – in der Regel „on demand“. Maßnahmen zur Einhaltung einer wie auch immer gearteten Vollständigkeitsbedingung sind nicht zu treffen, da alle benötigten Seiten der DB (welche die aktuelle Prädikatsextension enthalten) über die ausgewählten Zugriffspfade erreicht werden. DB-Aktualisierungen erfordern keine besonderen Vorkehrungen, da jede Seite höchstens einmal in ihrer aktuellen Version im DB-Puffer ist und eine zurückzuschreibende Seite die alte Seitenversion auf dem Externspeicher ersetzt. Unabhängig davon, ob eine aktualisierte Seite schon auf ihren sicheren Platz zurückgeschrieben wurde, ist sie nach Commit der ändernden Transaktion sofort allen anderen Transaktionen zugänglich. Der DB-Puffer stellt also immer den jüngsten Zustand einer Seite zur Verfügung. Da keine Replikate im (zentralisierten) DBS existieren, ergeben sich so keine Probleme mit der Datenkonsistenz – es wird immer die höchste Qualitätsstufe des *jüngsten transaktionskonsistenten Zustands* geboten.

2.2 Unterschiede bei DB-Caching

DB-Caching erfolgt in einem FE-DBS, das durch ein vollständiges DBS[2], typischerweise artgleich zum BE-DBS, verkörpert wird, also insbesondere auch einen eigenen DB-Puffer und eigene Externspeicher besitzt. Für die zu verwaltenden Daten werden Speicherungsstrukturen und Zugriffspfade angelegt, die bei Speichermangel im DB-Puffer auf den Externspeicher verdrängt werden können. Da die Daten beim DB-Caching prinzipiell Kopien (von Teilmengen) von BE-Daten darstellen, sind keine aufwendigen Sicherungsverfahren erforderlich. Es kann periodisch immer wieder mit einem leeren Cache begonnen werden. Das wäre beispielsweise eine einfache Möglichkeit, den DB-Cache zu „reinigen“, um ihn dann mit aktuellen Daten zu füllen.

2. Im Gegensatz zur Verwaltung eines DB-Puffers wird hier die volle Anfragefunktionalität (Suche, übersetzung und Optimierung) benötigt. Für die Modifikation der Cache-Inhalte sind außerdem Synchronisationsmaßnahmen erforderlich, so dass zunächst nur auf Logging/Recovery-Funktionalität verzichtet werden könnte, es sei denn, transaktionsgeschützte Änderungen sollen im Cache vorgenommen werden. Lediglich bei sehr einfachen Verfahren würde der Einsatz von DBS mit stark reduzierter Funktionalität genügen. Das würde jedoch bedeuten, dass mehrere DBS-Versionen zu warten und weiterzuentwickeln sind.

Im FE-DBS sollen vollständige SQL-Anfragen oder abgrenzbare Teilanfragen daraus (im Folgenden sprechen wir einfach nur von Anfragen) vor Ort abgewickelt werden. Die Gründe dafür liegen in erster Linie im Leistungsverhalten, um beispielsweise Cursor-Operationen (ONC, open/next/close) über langsame Übertragungskanäle oder weite Entfernungen zu minimieren. Andere Systemaspekte finden sich in der verbesserten Skalierbarkeit des Gesamtsystems und einer höheren Verfügbarkeit der Daten. Dabei läuft die lokale Verarbeitung genauso ab, wie sie in Abschnitt 2.1 detailliert wurde, während Zugriffe auf die BE-DB soweit wie möglich vermieden werden sollen. Der große Unterschied liegt darin, dass der Inhalt der FE-DB die Auswertung von Anfragen erlauben muss. Das bedeutet, das Verfahren muss dafür sorgen, dass bei der Verarbeitung *alle* für das betreffende SQL-Prädikat erforderlichen Daten im FE-DBS vorliegen (*Vollständigkeitsbedingung*). Anderenfalls ist das SQL-Prädikat zur Verarbeitung an das BE-DBS zu schicken. Im BE-DBS ist diese Bedingung automatisch erfüllt, da es ja zu jedem Zeitpunkt die gesamte DB verwaltet.

Wir gehen zur vereinfachten Darstellung des Verfahrens davon aus, dass alle Datenänderungen (zunächst) im BE-DBS durchgeführt werden. Durch ein Subskriptionsverfahren werden solche Änderungen, die Daten in den FE-DBs betreffen, automatisch verschickt, so dass diese Änderungen spätestens nach einer Zeitspanne δ ihre Kopien in den Caches erreichen und diese wieder konsistent mit dem BE-Zustand machen. Verlangt eine Anwendung höchste Datenkonsistenz, werden ihre Anfragen direkt an das BE-DBS weitergeleitet.

Die Anforderungen an das Verarbeitungskonzept im FE sind abhängig vom Datenmodell. Ist beispielsweise eine objektorientierte Schnittstelle vorgesehen, die nur Zugriffe über OIDs unterstützt, lässt sich die Erfüllung der *Vollständigkeitsbedingung* in einfacher Weise feststellen. Die große Herausforderung rührt also von deklarativen und mengenorientierten Sprachen her. An dieser Stelle ist DB-Caching vergleichbar mit der Client-seitigen Objektpufferverwaltung von objektorientierten DBS (OODBS) und den Caches in Persistenz-Frameworks für objektorientierte Programmiersprachen [14]. Solange die Zugriffsschnittstelle nur Objekte über eine OID anfordern kann – etwa bei Page- oder Object-Server-Architekturen [7] – sind bei der Cache-Verwaltung keine Korrektheits- oder Vollständigkeitsprobleme zu meistern. Ungleich schwieriger wird die Situation bei der Query-Server-Architektur. In [14] wird dazu noch

die Unterscheidung von 1-mengen- und n-mengenorientierten Anforderungen getroffen, wobei erstere einfachen Selektions- und letztere Verbundprädikaten entsprechen. Allerdings ist bei diesen Architekturen oft vorab Verarbeitungswissen vorhanden, das zum Prefetching von so genannten Verarbeitungskontexten [7] genutzt werden kann.

Als weitere Anforderung sollte es möglich sein, Daten vom BE-DBS dynamisch anzufordern und diese in den DB-Cache einzulagern. Weiterhin sollte das Verfahren mit minimalem Spezifikationsaufwand auskommen und adaptiv sein, um automatisch auf veränderte Anfragelasten reagieren zu können. Beispielsweise sollten Mengen von Datensätzen im DB-Cache invalidiert oder ersetzt werden, wenn das zugehörige SQL-Prädikat längere Zeit nicht mehr angesprochen wurde. Nur so lässt sich die Effizienz des Verfahrens aufrechterhalten, da anderenfalls ständig auch nicht mehr benötigte Datenkopien im Cache konsistent zu halten wären. Diese Anforderungen leuchten unmittelbar ein, wenn DB-Caches nicht nur als Middleware-Komponenten zentraler Unternehmensinfrastrukturen, sondern auch in entfernten Datenzentren und in Edge-Servern von CDNs eingesetzt werden sollen. Falls bei einer potenziell großen Anzahl[3] von DB-Caches jedesmal eine zeit-, orts- und lastabhängige *deklarative Spezifikation* der Cache-Inhalte erforderlich ist, verliert ein solches System schnell seine Administrierbarkeit.

2.3 Was ist bei DB-Caching zu entscheiden?

Während bei der DB-Pufferverwaltung kaum Vorkehrungen für den DB-Betrieb zu treffen sind – bestenfalls einige Überlegungen zur Wahl und Optimierung eines anwendungsspezifischen Ersetzungsalgorithmus –, sind beim DB-Caching viele Fragen und Einstellungen zu klären, bevor der Betrieb aufgenommen werden kann:

- Wozu soll der Cache genutzt werden?
  Wie sich noch zeigen wird, ist eine genaue Typisierung der Anfragen, die im Cache ausgewertet werden können, erforderlich. Dazu sind die Anwendungen zu analysieren, um bestimmte Typen von SQL-Prädikaten festzulegen. Werden zu viele Prädikate ausgewählt, wird möglicherweise wegen der Vollständigkeitsbedingung (fast) die ganze DB in den DB-Cache wandern. Sollen etwa bestimmte Data-Mining-Operationen im Cache abgewickelt werden, sind spezielle, darauf zugeschnittene Prädikate (z. B. für n:m-Verbunde) zu definieren.
- Wie wird der Cache-Inhalt spezifiziert?
  Am einfachsten wäre eine Liste von Anfragen, die im Cache beantwortet werden sollen. Diese Vorgehensweise ist i. Allg. zu statisch. Es ist darauf zu achten, dass eine Spezifikation des Cache-Inhalts möglichst wartungsfrei bleibt und sich einer veränderten Anfragelast anpasst. Dies lässt sich beispielsweise erreichen, indem nur bestimmte Anfragetypen spezifiziert werden, die wiederum nur bestimmte Tabellen/Sichten betreffen. Eine andere Form der Spezifikation bezieht sich auf zusammengehörige Tabellen – so genannte Cache-Groups –, deren Inhalte dynamisch durch parametrisierte Cache-Constraints bestimmt werden.
- Was soll im Cache gehalten werden?
  Werden zur Spezifikation des Cache-Inhaltes Anfragen vorgegeben, sind deren Ergebnisse – also einzelne unabhängige Tabellen und Sichten – zu speichern (query result caching). Andererseits können zusammenhängende Datensammlungen (abhängige, über Beziehungen verknüpfte Tabellen) im Cache gehalten werden, deren Inhalte die Auswertung von Anfragen mit gewissen Prädikatstypen – beispielsweise einfachen Gleichheitsprädikaten mit Verbunden – ermöglichen.
- Wann werden die Daten in den Cache geladen?
  Entsprechend der Inhaltsspezifikation könnten die Daten vor Beginn jeglicher Anfrageverarbeitung in den Cache geladen werden. Diese Vorgehensweise ist statisch und wird in [2] als „deklarativ“ bezeichnet. Eine Verbesserung wird erreicht, wenn der Cache erst nach einer Phase der Bedarfsanalyse gefüllt wird. Anzustreben ist natürlich ein automatisches und dynamisches Laden, d. h. bei Nachfrage spezifizierter Daten ist der Cache „on demand“ zu laden (in der Hoffnung, dass diese Daten in naher Zukunft wieder nachgefragt werden).
- Sind überlappende Daten im Cache zugelassen?
  Die disjunkte Speicherung überlappender Tabellen oder Sichten provoziert neben dem Caching Replikation, d. h., die Wartung wird aufwändiger und es können unterschiedliche Änderungszustände im Cache auftre-

3. Akamai betreibt Netze, die bereits heute fast 15000 Knoten (Edge Caching Servers) umfassen [2].

ten. Beim Caching von Anfrageergebnissen lässt sich dieses Problem nicht immer vermeiden (siehe DBProxy-Ansatz), da sich die Ergebnismengen von Selektions- und Verbundanfragen nicht immer in derselben Cache-Tabelle unterbringen lassen. Andererseits legt Anwendungstransparenz nahe, für jede BE-Tabelle höchstens eine FE-Tabelle (pro FE-DBS) anzulegen.

- Wann werden die Daten im Cache gewartet?
  Das einfachste Verfahren verzichtet auf Aktualisierung der im Cache gehaltenen Prädikat-Extensionen. Dabei könnte man vorsehen, dass eine eingelagerte Extension nach einem vorgegebenen Zeitintervall automatisch aus dem Cache entfernt wird, unabhängig von ihrer zwischenzeitlichen Nutzung. Eine spätere Prädikatreferenz kann dann zum erneuten Laden dieser Extension aus der aktuellen BE-DB führen.
  Ein adaptives Cache-Verhalten setzt jedoch die Möglichkeit voraus, Prädikatextensionen dynamisch zu warten. Eine solche Wartung impliziert, dass der Cache im Hinblick auf zwischenzeitlich erfolgte Änderungen in der BE-DB (mit Verzögerung) aktualisiert wird. Das bedeutet, dass typischerweise mengenorientierte Änderungen (Einfügen, Modifizieren und Löschen von Sätzen) abhängig von den im Cache verwalteten Prädikaten in deren Extensionen nachzuvollziehen sind. Die zeitgleiche Aktualisierung im BE und dem FE, über das die transaktionale Änderung ausgelöst wird, würde verteilten Transaktionsschutz mit einem 2PC-Protokoll [8] erfordern, was in der Regel zu teuer ist und trotz des hohen Synchronisationsaufwandes keine BE-DB-Konsistenz in *allen* FE-DBs garantieren kann. Denn auch bei dieser Vorgehensweise müssten Änderungen in anderen FE-DBs, deren Prädikatextensionen mit der geänderten Datenmenge überlappen, nachgezogen werden – notwendigerweise mit einem gewissen Zeitverzug. Deshalb ist es ein vernünftiges Ziel, das Nachziehen von Änderungen in den FE-DBs innerhalb einer Zeitspanne δ zu garantieren. Wenn diese Zeitspanne nicht explizit festgelegt wird und die Aktualisierung nur irgendwann erfolgt, ist das Verfahren zwar kostengünstiger, die zu garantierende Datenkonsistenz wird jedoch immer schwächer und tendiert zu „beliebig“.
- Wann werden Prädikatextensionen im Cache ersetzt bzw. invalidiert?
  Speichermangel als Grund für die Invalidierung bzw. Ersetzung von Prädikaten und deren Extensionen tritt nur dann auf, wenn ein FE-DBS mit einer festen Puffergröße betrieben wird und Überläufe oder Auslagerungen auf den Externspeicher vermieden werden sollen. Erlaubt ein Verfahren keine Ersetzung bzw. Invalidierung, trägt dies zwar zur Vereinfachung der Implementierung bei, ist im Betrieb jedoch nicht kosteneffektiv, da dann ständig auch nicht mehr genutzte Daten im Cache aktualisiert werden. Als adaptiv kann man ein solches Verfahren nicht bezeichnen.
  Ein gutes Verfahren sollte Prädikate und deren Extensionen nach Ablauf eines Zeitintervalls ohne Nutzung – unabhängig vom gewählten Wartungsverfahren – automatisch invalidieren und aus dem Cache entfernen.

Bei allen Wartungs- und Ersetzungsoperationen darf auch bei Prädikaten mit überlappenden Satzmengen die *Vollständigkeitsbedingung* nicht verletzt werden. Diese Forderung ist i. Allg. sehr schwer zu erfüllen und führt bei konkreten Verfahren notwendigerweise zur Einschränkung der Komplexität von zugelassenen Prädikaten.

Jetzt, nachdem durch diese Diskussion allgemeiner Eigenschaften von DB-Caching die Problemstellung genauer umrissen ist, konzentrieren wir uns im Wesentlichen auf die Darstellung zweier unterschiedlicher Ansätze, replikationsfreier Sichten sowie Cache-Groups, um konkrete Lösungen im Detail zu studieren. Wir hoffen dabei Hinweise auf den „gesamten“ Lösungsraum für die skizzierte Aufgabenstellung zu finden, um unsere Aussagen dann später verallgemeinern zu können.

3 Existierende Ansätze für das DB-Caching

Einfache Ansätze zum DB-Caching sind nicht adaptiv, müssen weitgehend vorgeplant werden und verlangen ständige Überwachung durch die DB-Administration. Typischerweise werden die im Cache zu haltenden Daten *vorab deklarativ spezifiziert* und geladen; Spezifikationsänderungen erfolgen manuell, d. h., Leistungsverschlechterungen im Cache-Verhalten müssen von außen erkannt und behoben werden, indem man etwa neue Cache-Objekte spezifiziert und mit ihnen alte ersetzt. Wenn alle Cache-Objekte vorab genau spezifiziert sind und ihr Einlagern sowie ihre Aktualisierung detailliert festgelegt ist, führt diese Art von Caching auf ähnliche Probleme wie bei Replikationsverfahren, ggf. mit periodischer Aktualisierung (Schnappschüsse

mit vorgegebenen Refresh-Intervallen). Als Hauptunterschied erfolgt bei der Replikation das Anfrage-Routing durch das Anwendungsprogramm, da der Ort eines Replikats frei wählbar ist. Dagegen bleibt die Nutzung eines Objektes im Cache unsichtbar für die Anwendung, da der Cache sich im von der Anwendung benutzten Aufrufweg befindet.

Alle im Folgenden diskutierten Ansätze wollen eine Überwachung durch die DB-Administration weitestgehend vermeiden; sie wollen dynamisch und adaptiv sein. Offensichtlich umfassen sie als einfachsten Grenzfall *deklaratives Caching*, so dass dieses nicht weiter untersucht werden muss.

3.1 Materialisierte Sichten

Offensichtliche Verfahren zum DB-Caching beruhen auf der Nutzung von replizierten Tabellen oder materialisierten Sichten [4, 6]. Jede Tabelle in der FE-DB entspricht einer BE-Tabelle oder einer Sicht, die durch eine auf Selektion und Projektion beschränkte Anfrage auf eine BE-Tabelle spezifiziert worden ist. Deshalb bezeichnet man diese Vorgehensweise auch als Caching von Anfrageergebnissen (query result caching). Eine solche Sicht gewährleistet die erforderliche Vollständigkeitsbedingung für alle nachfolgenden Anfragen, die durch die Sichtdefinition subsumiert werden, d. h., die mithilfe des Anfrageergebnisses im Cache vollständig beantwortet werden können. Da jede Sicht als separate FE-Tabelle gespeichert wird, besteht bei überlappenden Sichten das Problem, dass im Cache Replikation auftritt. Neben einem erhöhten Speicherplatzbedarf führt dieser einfache Ansatz auf erhebliche Konsistenzprobleme, wenn die einzelnen Sichten im Cache zueinander konsistent gehalten werden sollen.

Als erstes anspruchsvolleres Verfahren diskutieren wir DBProxy[4], das Sichten im DB-Cache dynamisch verwaltet und dabei Replikation vermeiden will.

3.2 Replikationsfreie Sichten

Indem man den Zusammenhang zwischen verschiedenen Sichten, etwa deren Bezug auf gleiche Tabellen, ausnutzt, lassen sich diese Sichten in gemeinsam genutzten Strukturen im Cache materialisieren. Ein konkretes Verfahren zur (zumindest in der Zielvorstellung) replikationsfreien Sichtenüberlagerung stellt DBProxy dar [3]. Es nimmt die Ergebnisse von Anfragen vollständig in den Cache auf, speichert sie replikationsfrei und versucht aus der Menge der gespeicherten Anfrageergebnisse weitere Anfragen zu beantworten.

3.2.1 Cache-Inhalt

Wir beschränken uns im Folgenden auf einfache Selektions- und Projektionsanfragen, die jeweils nur eine BE-Tabelle betreffen. DBProxy hat zwar das Ziel, auch mit Verbundanfragen umgehen zu können, allerdings sind in diesem Bereich noch nicht mehr Konzepte vorhanden als eine primitive Erweiterung des Falles mit einer Tabelle.

Anfragen, die eine bestimmte BE-Tabelle betreffen,[5] werden so umgeschrieben, dass ihre Ergebnisse den Primärschlüssel enthalten. Diese Ergebnisse können dann in einer gemeinsamen FE-Tabelle abgelegt werden; der Primärschlüssel verhindert Duplikate. Unterscheiden sich verschiedene Anfrageergebnisse, die aus derselben BE-Tabelle stammen, in der Menge der zurückgelieferten Spalten, machen Nullwerte die Ergebnisse zur Speicherung syntaktisch kompatibel. Das Verfahren, nach dem Cache-Inhalte zur Beantwortung neuer Anfragen herangezogen werden, verhindert, dass diese „falschen“ Nullwerte mit Nullwerten in den Anfrageergebnissen verwechselt werden.

Abb. 3 zeigt den Inhalt einer Kundentabelle C (als BE-C im BE gespeichert) und den Cache-Zustand nach zwei Anfragen Q_1 und Q_2 gegen C. Dabei wird unterstellt, dass die Ergebnisse beider Anfragen in einer Tabelle FE-C im Cache gehalten werden sollen. Um Überlappungen in den Sätzen erkennen zu können, wird die Ausgabeliste jeder Anfrage automatisch um den Primärschlüssel ergänzt. Ausgehend von einem leeren Cache wird deshalb Q_1 um den Primärschlüssel Cno erweitert. FE-C wird mit zunächst drei Spalten angelegt, um das (überstrichene) Anfrageergebnis aufzunehmen. Die nachfolgende Anfrage Q_2 erfährt eine ähnliche Behandlung; FE-C wird um die neue Spalte Cid erweitert, damit das (unterstrichene) Ergebnis eingefügt werden kann. Dabei wird zum einen der schon vorhandene

4. Alle betrachteten Forschungsansätze sind im Moment nur prototypisch realisiert und beantworten nicht alle im Abschnitt 2.3 adressierten Fragen. Insbesondere werden bisher Verfahren zum Ersetzen bzw. Invalidieren von Daten im Cache ausgespart.

5. In der erwähnten Erweiterung müssen sich diese Anfragen auf eine bestimmte Menge von BE-Tabellen mit von Fall zu Fall gleichem Verbundprädikat beziehen.

BE-C	Cno	CType	CLoc	Cid
	1	silver	SF	NULL
	2	silver	LA	a
	3	bronze	SJ	b
	4	NULL	NY	e
	5	gold	SJ	f
	6	gold	SF	g
	7	gold	NY	h

$\overline{Q}_1$: select CType, CLoc from C where CType = gold

$\underline{Q}_2$: select CLoc, Cid from C where Cid between b and f

FE-C	Cno	CType	CLoc	Cid
	$\underline{3}$	*NULL*	$\underline{SJ}$	$\underline{b}$
	$\underline{4}$	*NULL*	$\underline{NY}$	$\underline{e}$
	$\underline{\overline{5}}$	$\overline{gold}$	$\underline{\overline{SJ}}$	$\underline{f}$
	$\overline{6}$	$\overline{gold}$	$\overline{SF}$	*NULL*
	$\overline{7}$	$\overline{gold}$	$\overline{NY}$	*NULL*

Abb. 3: Überlappende Speicherung mit falschen Nullwerten. Die Spalte Cid kommt erst durch Q_2 hinzu.

Satz 5 durch den Wert von Cid ergänzt; in allen anderen Sätzen werden für unbekannte Spalten Nullwerte eingesetzt.

Um eine häufige Erweiterung der FE-Tabellen um neu auftretende Spalten zu vermeiden, bietet sich eine einfache Optimierung an: Ist die Menge der in einer erwarteten Anfragelast benutzten Spalten bekannt, können die FE-Tabellen von vornherein passend angelegt werden. Erweitert man dann jede Anfrage (hier Q_1 und Q_2) um alle fehlenden Spalten, vergrößert sich mit wenig Mehraufwand die Menge der potentiell aus dem Cache beantwortbaren Anfragen (Abb. 4). Jedoch ist zu beachten, dass alle Spalten konsistent gehalten werden müssen, also zusätzlichen Wartungsaufwand einführen, der sich nicht amortisiert, wenn diese zusätzlichen Spalten durch Anfragen nicht genutzt werden.

3.2.2 Auswertung von Anfragen im Cache

Um eine neue Anfrage korrekt beantworten zu können, muss der Cache entscheiden, ob deren Ergebnis in der Vereinigung aller bisher im Cache vorhandenen Ergebnisse enthalten ist, d. h., ob der Cache die Vollständigkeitsbedin-

$\overline{Q}_1$, $\underline{Q}_2$ wie in Abb. 3

FE-C	Cno	CType	CLoc	Cid
	$\underline{3}$	$\underline{bronze}$	$\underline{SJ}$	$\underline{b}$
	$\underline{4}$	$\underline{NULL}$	$\underline{NY}$	$\underline{e}$
	$\underline{\overline{5}}$	$\overline{gold}$	$\underline{\overline{SJ}}$	$\underline{\overline{f}}$
	$\overline{6}$	$\overline{gold}$	$\overline{SF}$	$\overline{g}$
	$\overline{7}$	$\overline{gold}$	$\overline{NY}$	$\overline{h}$

Abb. 4: Optimierung durch Vorab-Definition einer umfassenden Tabelle

gung für die neue Anfrage erfüllt. Bei diesem Subsumptionstest ist nicht nur sicherzustellen, dass alle benötigten Sätze vorhanden sind, sondern auch, dass diese alle von der neuen Anfrage benötigten Spalten umfassen. Dazu gehören neben den Projektionsspalten auch alle, die zur Durchführung der Selektion benötigt werden. Im aktuellen Vorschlag [3] wird zunächst nach diesem Kriterium eine Vorauswahl unter den gespeicherten Anfragen getroffen; die resultierenden Q_i sind Kandidaten zur Abdeckung der neuen Anfrage Q_B. Ist dann ein automatischer Beweis erfolgreich, dass das Where-Prädikat von Q_B aus der Disjunktion der Where-Prädikate der Q_i logisch folgt, kann Q_B sicher im Cache ausgewertet werden [13], da alle benötigten Sätze vorhanden sind. Alle Spaltenwerte, die man von diesen Sätzen verwendet, sind wegen der Vorauswahl echt. Auf diese Weise gelangen nie falsche Nullen in das Anfrageergebnis.

3.2.3 Aktualisierung

Treten Änderungen an den Tabellen im BE auf, müssen diese in den FE-Tabellen nachvollzogen werden, sofern sie die Extension eines der dort vorgehaltenen Prädikate betreffen. Um diese Entscheidung treffen zu können, verwaltet das BE eine Subskriptionsliste und gibt Änderungen möglicherweise an mehrere betroffene FEs weiter.

Betrachten wir erneut den Zustand des Caches in Abb. 4: Q_1 und Q_2 werden im Cache gehalten; ihre derzeitigen Extensionen befinden sich in FE-C. In der Subskriptionsliste des BE sind Q_1 und Q_2 mit diesem FE assoziiert. Zwei beispielhaft in BE-C eingefügte Sätze müssen auch in FE-C aufgenommen werden: Satz 9 erfüllt das Prädikat Q_2, Satz 8 sowohl Q_1 als auch Q_2 (Abb. 5).

Soll nun das Prädikat Q_2 aus dem Cache entfernt werden, kann dies nicht isoliert von anderen Prädikaten gesche-

FE-C	Cno	CType	CLoc	Cid
	3	bronze	SJ	b
	4	NULL	NY	e
	5	gold	SJ	f
	6	gold	SF	g
	7	gold	NY	h
	8	gold	SF	d
	9	bronze	SJ	c

Abb. 5: Einfügung durch Konsistenzprotokoll

hen: Gemeinsam genutzte Sätze (hier Sätze 5 und 8) müssen im Cache verbleiben, können aber von nicht mehr benötigten Attributen befreit werden[6]. Sätze 3, 4 und 9 sind allein wegen Q2 in FE-C und werden deshalb gelöscht.

3.3 Cache-Groups

Cache-Groups wurden unseres Wissens erstmalig im System TimesTen [17] eingesetzt. Im Projekt DBCache [2] wurde ihre Flexibilität verbessert und erste Maßnahmen zur Adaptivität hinzugefügt. Cache-Groups sind Vertreter einer Richtung von DB-Caching-Verfahren, die auf der Spezifikation von *Cache-Constraints* basieren: Im FE-DBS arbeiten diese mit Cache-Tabellen, die aus Gründen der Anwendungstransparenz in ihrem Schema den BE-Tabellen entsprechen und eine Teilmenge der Sätze aus diesen Tabellen enthalten; die Tabellen- und Spaltennamen sind der Einfachheit halber identisch.[7] Dabei werden nicht direkt Anfrageergebnisse übernommen, sondern der Zusammenhang zwischen Tabellen einer Cache-Group – und damit implizit der Cache-Inhalt – wird durch Constraints definiert.

3.3.1 Spezifikation des Cache-Inhalts

Der Administrator bestimmt die Teilmenge der BE-Tabellen, für die strukturgleiche Cache-Tabellen im FE-DBS angelegt werden. Zwei Arten von *Cache-Constraints* spezifizieren den Inhalt von und die Abhängigkeiten zwischen Cache-Tabellen: Cache-Keys lösen das Einlagern von Sätzen in den Cache aus; RCCs (Referential Cache Constraints) sorgen für das Nachladen der „Umgebung“ eines Satzes. Die *Bereichsvollständigkeit* von Spalten erlaubt (den Einstieg für) die Auswertung von Anfragen über Gleichheitsprädikate.

Der Ausgangspunkt zur Gewährleistung der Vollständigkeitsbedingung für ein Prädikat ist die Eigenschaft der Bereichsvollständigkeit (Domain Completeness, DC) von Spalten in FE-Tabellen.

- *Definition:* Eine Spalte c von FE-Tabelle S heißt genau dann *bereichsvollständig*, wenn gilt: Wann immer ein Wert x in S.c ist, sind alle Sätze aus $\sigma_{c = x}S_B$ in S.

Offensichtlich erlaubt diese DC-Eigenschaft die Auswertung des Gleichheitsprädikats $c = x$ im Cache.

Ein *RCC* macht eine Cache-Tabelle von einer anderen abhängig, indem er eine wertbasierte Beziehung zwischen zwei Spalten herstellt. Start- und Zielspalte eines RCCs müssen nicht notwendigerweise in zwei verschiedenen Tabellen liegen (sie können sogar identisch sein).

- *Definition:* Ein RCC $S.a \rightarrow T.b$ zwischen zwei Spalten[8] a und b von FE-Tabellen S und T ist erfüllt, wenn gilt: Wann immer ein Wert x in $S.a$ ist, sind alle Sätze aus $\sigma_{b = x}T_B$ in T.

Nach dieser Definition garantiert ein RCC für alle sich in der Startspalte befindenden Werte die Korrektheit von Gleichverbunden zwischen zwei Cache-Tabellen.

RCCs sorgen immer nur ausgehend von schon im Cache enthaltenen Sätzen für das Nachladen weiterer. Zum Füllen des Caches sind deshalb noch ausgezeichnete Spalten erforderlich, die als *Füllpunkte* fungieren, an denen also Werte mit ihren zugehörigen Sätzen in den Cache gelangen. Diese explizit zu spezifizierenden Spalten heißen *Cache-Keys* und bewirken bei der Referenz eines Wertes aus dieser Spalte das Laden aller Sätze, die diesen Wert tragen. Diese Referenzen können dabei von außen, aber auch von innen kommen: Einerseits werden Cache-Key-Werte (wie Werte aus bereichsvollständigen Spalten) in Anfragen benutzt, andererseits gelangen Sätze über RCCs in FE-Tabellen mit Cache-Keys. Solche Tabellen enthalten also stets zu jedem der Werte eines Cache-Keys alle zugehörigen BE-Sätze. Damit ist jede Cache-Key-Spalte *bereichsvollständig.* Auch ande-

6. Bei sorgfältiger Implementierung schadet es allerdings nicht, solche Attribute nicht auf Null zurückzusetzen: Wie falsche Nullwerte sind sie nicht mehr erreichbar.

7. Falls eine Unterscheidung erforderlich ist, benutzen wir die Präfixe FE und BE oder die Indizes F und B.

8. Die Werte beider Spalten $S.a$ und $T.b$ müssen verbundverträglich sein; anderenfalls erzeugt jeder Verbund entlang eines RCCs ein leeres Ergebnis. In SQL sind beide Spalten typischerweise auf demselben Wertebereich definiert.

re Spalten können diese Eigenschaft besitzen, vor allem Unique-Spalten – bei ihnen existiert ja pro Wert jeweils nur ein Satz. Zielspalten von RCCs sind demgegenüber nicht ohne weiteres bereichsvollständig: Der RCC stellt dort zwar stets die Vollständigkeit der aus der Quellspalte kommenden Werte sicher, aber auch auf anderen Wegen können Werte in diese Zielspalte gelangen.

Der zur Unterstützung eines einzelnen Anwendungsfalls nützliche Cache-Inhalt wird also in der Regel durch einen Cache-Key und eine Menge von RCCs beschrieben. Im Hinblick auf die so betroffene und verknüpfte Gruppe von Tabellen – die Cache-Key-Tabelle dient als Wurzeltabelle, über die RCCs sind weitere Tabellen erreichbar – heißt diese Konfiguration *Cache-Group*. Eine Cache-Group lässt sich als parametrisiertes Prädikat auffassen: Bei Festlegung eines konkreten Cache-Key-Werts als Parameter entsteht ein Prädikat, dessen Extension entsprechend der Vollständigkeitsbedingung vollständig im Cache gehalten wird. Bei einer einfachen Cache-Group $S \rightarrow T$ mit Wurzeltabelle S und erreichbarer Tabelle T, einem Cache-Key S.c und einer RCC $S.a \rightarrow T.b$ sind beispielsweise nach einer Referenz S.c = x alle solche Sätze der BE-DB im Cache, die eine vollständige Auswertung des Prädikats (S.c = x and S.a = T.b) erlauben. Ein erstmals referenzierter Cache-Key-Wert dient folglich als eine Art Indikator dafür, dass in nächster Zukunft Referenzlokalität auf der von ihm abhängigen Datenmenge erwartet wird.

Die Spezifikation einer Cache-Group selbst ist statisch und passt sich nicht an veränderte Umgebungsbedingungen an. Seine Adaptivität erhält das Verfahren allein durch seine am Bedarf orientierte Füllung und Wartung des Caches in den Schranken dieser Spezifikation.

3.3.2 Füllverhalten

Um das Verhalten von Cache-Keys zu verdeutlichen, betrachten wir in Abb. 6 die schon bekannte BE-Kundentabelle BE-C sowie eine zugehörige FE-Tabelle FE-C mit nur einem Cache-Key CType[9]. Cno und Cid seien Spalten vom Typ Unique (U), CType und CLoc vom Typ Nonunique (NU). Ausgehend von einer leeren Tabelle FE-C führt eine Referenz mit dem Prädikat CType = gold zum Laden der Sätze 5 bis 7. Entsprechend werden alle silver-Sätze (1, 2) geladen, falls wir danach CType = silver referenzieren. Wegen der Bereichsvollständigkeit von CType und Cid könnten Anfragen nach goldenen und silbernen Kunden jetzt aus dem Cache beantwortet werden. Das gleiche gilt für eine Anfrage nach Cid = *g*, allerdings würde diese Anfrage niemals zum Laden von Satz 6 führen, da Cid kein Cache-Key ist.

BE-C	Cno	CType	CLoc	Cid
	1	silver	SF	NULL
	2	silver	LA	a
	3	bronze	SJ	b
	4	NULL	NY	e
	5	gold	SJ	f
	6	gold	SF	g
	7	gold	NY	h

FE-C	Cno	CType	CLoc	Cid
	5	gold	SJ	f
	6	gold	SF	g
	7	gold	NY	h
	1	silver	SF	NULL
	2	silver	LA	a

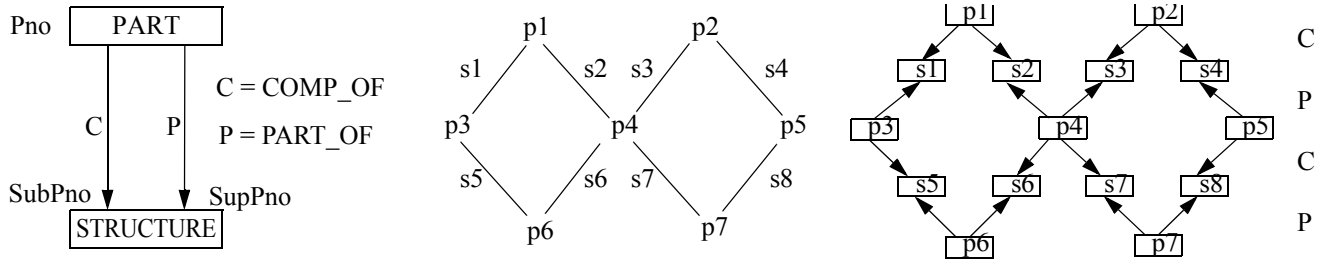
Abb. 6: Einsatz von Cache-Keys

Wir untersuchen nun das Zusammenspiel mehrerer Cache-Keys, indem wir FE-C neben CType mit einem weiteren NU-Cache-Key CLoc ausstatten (Abb. 7). Eine Referenz CType = bronze führt dann (bei zunächst leerem Cache) zum Laden des Satzes 3. Dies erfordert ein Nachladen aller weiteren Sätze mit CLoc = SJ, um der Bereichsvollständigkeit des zweiten Cache-Keys zu genügen. Dieser eine Satz 5 führt allerdings einen neuen CType-Wert in die

FE-C	Cno	CType	CLoc	Cid
	3	bronze	SJ	b
	5	gold	SJ	f
	6	gold	SF	g
	7	gold	NY	h
	1	silver	SF	NULL
	4	NULL	NY	e
	2	silver	LA	a

Abb. 7: Notwendigkeit der Cache-Key-Regel

9. CType wurde hier der Anschaulichkeit halber gewählt. In praktischen Anwendungen ist die Kardinalität oder Werteverteilung einer Spalte kritisch zu prüfen, bevor sie als Cache-Key spezifiziert wird. Im Allgemeinen muss ein Verfahren vorgesehen werden, um bei der Füllung des Cache einzelne Werte eines Cache-Key auszuschließen (Stoppwort- oder Negativliste) oder nur bestimmte Werte (Positivliste) zu berücksichtigen.



a) BE-Schema b) Knoten-/Kantenbezeichnungen im GG c) GG als Graph mit PS/FS-Beziehungen

Abb. 8: Stückliste als Schema und mit Ausprägungen

FE-Tabelle ein, so dass der Cache-Key CType erneut aktiv werden muss und das Nachladen der Sätze 6 und 7 erzwingt. Dieses Ping-Pong-Spiel setzt sich fort, bis keine neuen Cache-Key-Werte mehr in FE-C auftauchen. In unserem Beispiel wird dabei die gesamte BE-Tabelle in den Cache geladen. Um ein derart unkontrolliertes Füllverhalten zu vermeiden, verbieten wir die Existenz von mehr als einem NU-Cache-Key pro Tabelle (*Cache-Key-Regel*). U-Cache-Keys können in beliebiger Anzahl toleriert werden; sie führen nie zum Nachladen weiterer Sätze.

Wir gehen nun zu Cache-Groups über, die mehr als eine Tabelle enthalten. Für die detailliertere Betrachtung verschiedener Konfigurationen ist es nützlich, RCCs nach Eigenschaften der beteiligten Spalten zu klassifizieren: Bei der Unterscheidung von Unique- und Nonunique-Spalten ergeben sich so folgende vier RCC-Typen:

- $U \rightarrow U$
- $U \rightarrow NU$, z. B. Primär-/Fremdschlüssel-Beziehung oder Member-Constraint (MC)
- $NU \rightarrow U$, z. B. Fremd-/Primärschlüssel-Beziehung oder Owner-Constraint (OC)
- $NU \rightarrow NU$ (Cross-Constraint, XC).

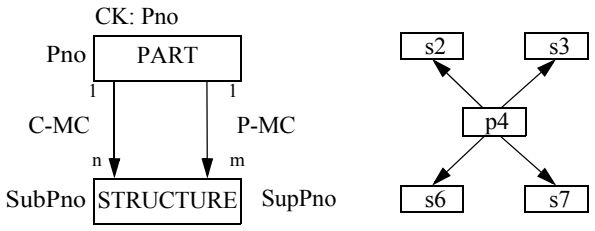
In Abb. 8a ist das typische Schema einer Stückliste gegeben, bestehend aus Teile-Relation PART sowie Struktur-Relation STRUCTURE mit Fremdschlüsseln zum Verweis auf übergeordnete (P) und untergeordnete (C) Teile. Wir betrachten eine konkrete Stückliste in ihrer Darstellung als Gozinto-Graph GG (Abb. 8b) und abgebildet auf die Primär-/Fremdschlüssel-Beziehungen unseres Schemas (Abb. 8c). Der Übersichtlichkeit halber stellen wir diese Belegung unserer BE-Tabellen auch als Graph dar. Wir spezifizieren nun im Folgenden verschiedene Cache-Groups über dieser BE-Instanz und konzentrieren uns bei unserer Analyse des Füllverhaltens auf das Zusammenspiel verschiedener RCCs (Abb. 9). Als Cache-Key nehmen wir durchgängig die Teilenummer Pno an.

Eine erste einfache Konfiguration (a) besteht aus zwei RCCs C-MC und P-MC, die dafür sorgen, dass mit jedem Teil die zugehörigen Struktur-Sätze in den Cache geladen werden: Eine Referenz von Pno = p4 führt über C-MC zum Nachladen von s6 und s7, über P-MC zum Nachladen von s2 und s3.

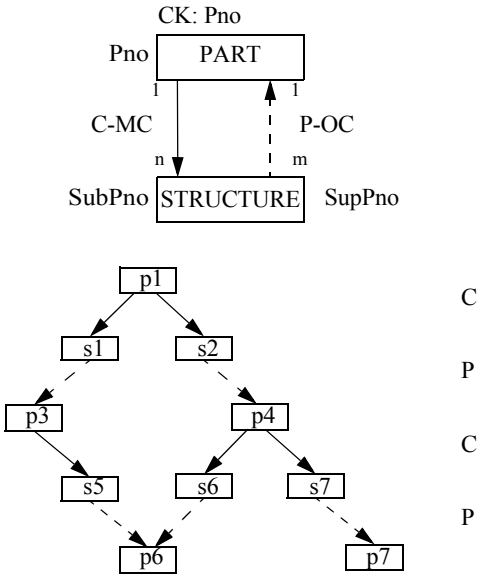
Ändern wir die Orientierung eines der beiden RCCs und machen so P-MC zu P-OC, entsteht ein RCC-Zyklus (b). Dies äußert sich direkt im Füllverhalten: Eine Referenz von Pno = p1 lädt zunächst p1, dann s1 und s2 via C-MC, dann p3 und p4 via P-OC, dann s5 bis s7 via C-MC und so fort. Wir erleben hier also ein rekursives Füllverhalten, ähnlich dem bei der Verletzung der Cache-Key-Regel. Dieses Verhalten kann noch extremere Züge annehmen, wenn Rekursion entlang mehrerer Wege möglich ist (c): Spezifizieren wir alle vier in diesem Beispiel sinnvollen RCCs, schreitet das Laden des Caches bei einer Referenz von Pno = p1 wie im vorangegangenen Beispiel voran, bis der Satz p4 erreicht wird. Von dort pflanzt sich der Effekt der Cache-Key-Referenz in alle Richtungen fort. Schließlich befindet sich die gesamte Stückliste im Cache.[10]

Zyklen in Cache-Groups können also ebenfalls ein unkontrolliertes und damit untragbares Füllverhalten hervorrufen. Schlimmstenfalls zieht rekursives Laden ganze BE-Tabellen in den Cache. Allerdings existieren auch „harmlo-

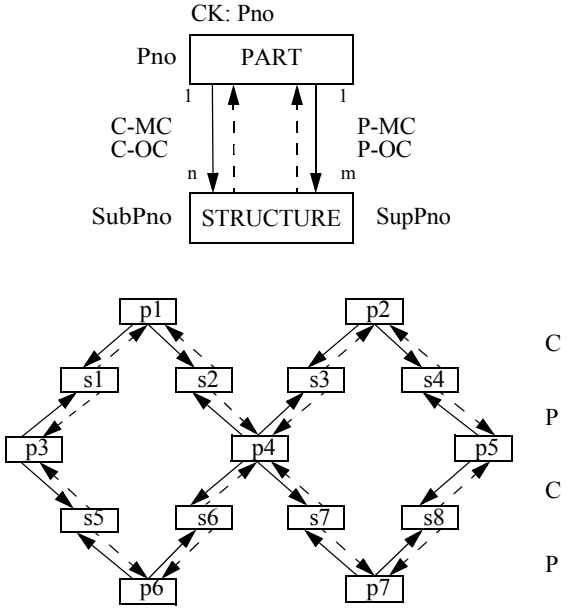
10. Diese Form des rekursiven Nachladens ist offensichtlich viel teurer als das sofortige Laden der ganzen Tabelle. Jedoch ist einer Referenz nicht „anzusehen“, ob sie das Laden der ganzen Tabelle erzwingt (bei in sich geschlossenen Referenzbeziehungen würden nur isolierte Bereiche nachgeladen werden). Deshalb würde das Laden ganzer Tabellen hier zu einem hybriden Verfahren (mit Full-Table Caching) führen, was wir nicht vertiefen wollen, da es eine andere Art der Cache-Aktualisierung voraussetzt.



a) Einfache Cache-Group für die Stückliste

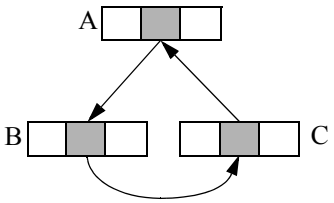


b) Cache-Group mit einfacher Rekursion

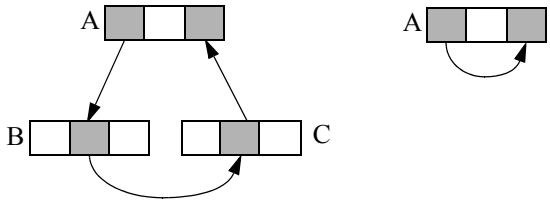


c) Cache-Group mit doppelter Rekursion

Abb. 9: Füllverhalten von Cache-Groups



a) homogener Zyklus



b) heterogene Zyklen

Abb. 10: Einschränkungen bei Cache-Groups

se“ Zyklen, beispielsweise in Abb. 10a. Bei diesem Zyklus sind bei jeder Tabelle Zielspalte des eingehenden RCC und Startspalte des ausgehenden RCC identisch. Somit wird immer derselbe Spaltenwert über den Zyklus propagiert; das Nachladen stoppt spätestens nach einem Umlauf. Solche Zyklen heißen *homogen;* alle Zyklen, bei denen beim Übergang von einem RCC zum nächsten irgendwo im Zyklus ein Spaltenwechsel auftritt, heißen dagegen *heterogen* (Abb. 10b).

Durch ein Verbot der „gefährlichen“ heterogenen Zyklen lässt sich also rekursives Laden vermeiden. Entsprechend gefährlich sind Kombinationen aus Zyklen und Cache-Keys, bei denen wie bei der Cache-Key-Regel zwei NU-Spalten involviert sind (Abb. 11).

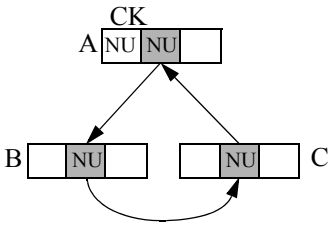


Abb. 11: Gefährliche Situation durch homogenen Zyklus und Cache-Key

3.3.3 Auswertung von Anfragen im Cache

Jede bereichsvollständige Spalte in einer der Cache-Tabellen unterstützt die Auswertung von Gleichheitsprädikaten der Form „Spalte = Wert“: Für jeden Wert dieser FE-Spalte ist sichergestellt, dass sich alle Sätze, die dem entsprechenden Gleichheitsprädikat genügen, im Cache befin-

den. Darüber hinaus können bereichsvollständige Spalten als *Einstiegspunkte* dienen, um die Eigenschaften der RCCs auszunutzen. Sobald durch eine Prüfanfrage (Probing) feststeht, dass sich der geforderte Wert in einer DC-Spalte befindet, können (Gleichheits-)Verbunde entlang den RCCs im Cache ausgeführt werden, ohne die Korrektheit der Ergebnisse zu beeinträchtigen. Für jede in einer Anfrage benutzte Tabelle, für die nicht auf diese Weise eine Cache-Tabelle eingesetzt werden kann, muss auf die BE-Tabelle zurückgegriffen werden. Technisch lässt sich dieses Problem durch Janus-Anfragepläne [2] meistern, die verteilt abgewickelt werden können (Federated Query Facility [9]).

3.3.4 Implizite DC-Spalten und RCCs

Da zur Entscheidung der Frage, welche Gleichheits- und Verbundsprädikate man prinzipiell im Cache auswerten kann, nur die DC-Spalten und die RCCs herangezogen werden, gibt es Raum für Optimierungen, indem man zusätzliche Spalten oder Beziehungen identifiziert, die dieselben hilfreichen Eigenschaften aufweisen, ohne explizit so definiert worden zu sein.

Wir betrachten einen Ausschnitt aus einer Cache-Group mit einer Cache-Tabelle B, die nur einen von der Tabelle A kommenden RCC des Typs MC besitzt, außerdem keine Cache-Keys (Abb. 12). Die Tabelle B wird also ausschließlich über diesen MC befüllt, so dass dessen Zielspalte immer die DC-Eigenschaft aufweist: Entweder ist die Spalte leer oder es kommen stets alle Sätze eines Wertes hinzu. Andersherum betrachtet muss zu jedem Satz t in B ein Owner-Satz in A existieren, der das Einfügen von t bewirkt hat. Da die Startspalte des MC zusätzlich unique ist, gilt also der umgekehrte RCC; selbst wird er jedoch nie zum Laden von neuen Sätzen führen.

Wir nennen diese Art der Bereichsvollständigkeit bzw. diese Art von RCCs *implizit*: Sie sind nicht durch Spezifikation von Cache-Key- oder Unique-Constraints (*explizite* DC) bzw. Spezifikation von RCCs (*explizite* RCCs) direkt gegeben, sondern entstehen aus dem Zusammenspiel dieser in konkreten Situationen.

Implizite DC-Spalten und RCCs sind nur für eine bestimmte Cache-Group-Konfiguration gültig und können verloren gehen, wenn diese geändert wird. Nehmen wir in unserem Beispiel eine Tabelle C und einen weiteren RCC hinzu, so dass B nun auch auf diesem Weg über eine andere

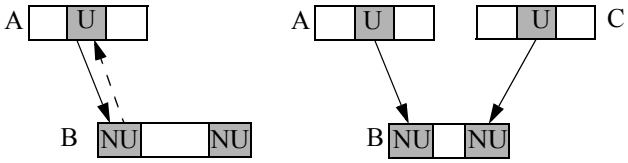


Abb. 12: Explizite und implizite Bereichsvollständigkeit

NU-Spalte gefüllt wird, hat die ehemals implizit bereichsvollständige Spalte diese Eigenschaft nicht mehr, und auch der RCC $B \rightarrow A$ gilt nicht (Abb. 12, rechts).

3.3.5 Aktualisierung

Wir illustrieren die Auswirkungen einer Änderung im BE auf die Cache-Tabellen, indem wir ausgehend vom Zustand in Abb. 6 die gleichen Sätze wie im Falle DBProxy in BE-C einfügen. Die Entscheidung, ob die Änderungen für ein bestimmtes FE relevant sind, muss natürlich auch hier anhand der dort vorgehaltenen Prädikate erfolgen. Allerdings sind im Falle von Cache-Groups diese Prädikate nur implizit gegeben (vgl. Abschnitt 3.3.1).

Satz 8 muss aufgrund seines CType-Wertes in die Cache-Tabelle FE-C aufgenommen werden und vervollständigt dort nur die Menge aller goldenen Kunden, die sich bereits im Cache befindet (Abb. 13). Für die Menge der bron-

FE-C	Cno	CType	CLoc	Cid
	5	gold	SJ	f
	6	gold	SF	g
	7	**bronze**	NY	h
	1	silver	SF	NULL
	2	silver	LA	a
	8	gold	SF	d
	3	bronze	SJ	b
	9	bronze	SJ	c

Abb. 13: Auswirkungen von Änderungen im BE

zenen Kunden gilt das nicht, so dass die Einfügung von Satz 9 im BE hier nicht im Cache wiederholt würde. Betrachten wir stattdessen eine Änderung von Satz 7 vom CType gold auf bronze. In der Konsequenz müssten alle anderen bronze-Sätze in den Cache folgen (hier der alte Satz 3 sowie der eben im BE hinzugekommene Satz 9): In jedem Fall ist die Vollständigkeitseigenschaft zu wahren, damit die Auswertung von Anfragen im Cache korrekt bleibt.

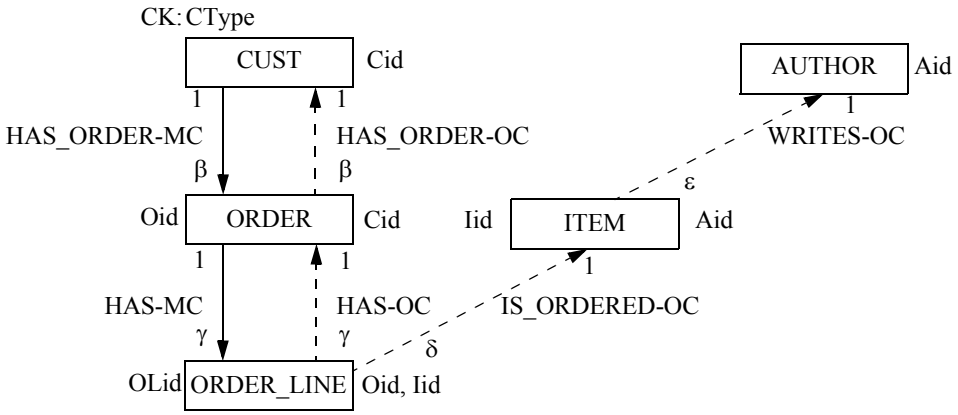


Abb. 14: Einsatzmöglichkeiten von Cache-Groups

3.3.6 Ein Anwendungsbeispiel für Cache-Groups

Ein Ausschnitt eines DB-Schemas für einen Web-Buchhändler (siehe [3]) könnte die in Abb. 14 gezeigten Tabellen CUST, ORDER, ORDER_LINE, ITEM und AUTHOR in der BE-DB umfassen. Die zugeordneten Attributnamen verweisen im Wesentlichen auf die Primär- und Fremdschlüssel, welche die existierenden relationalen Invarianten in diesem DB-Ausschnitt verkörpern. In der FE-DB können nun mit Bezug auf diese Tabellen (in einem wertbasierten Tabellenmodell) Cache-Constraints eingetragen werden, die eine (oder mehrere) Cache-Groups definieren. Um Platz zu sparen, beziehen wir uns bei allen unseren Betrachtungen auf Abb. 14 und erörtern dabei mehrere Cache-Groups mit wachsender Komplexität.

Cache-Group CG1 bestehe nur aus der FE-Tabelle CUST (kurz C) mit dem Cache-Key CType (NU) und dem Primärschlüssel Cid (U). CG1 als einfachst mögliche Cache-Group würde nur Anfragen über Gleichheitsprädikate $\text{CType} = x$ oder $\text{Cid} = y$ zu beantworten erlauben, wenn durch Probing der aktuelle Wert x oder y im Cache gefunden wird. Bei allen Beispielen skizzieren wir nur die Grundprädikate, die sich aus der Struktur einer Cache-Group ergeben. Natürlich können beliebige Einschränkungen mit „and“ angehängt werden.

Cache-Group CG2 entstehe durch Erweiterung von CG1 durch die FE-Tabelle ORDER (kurz O) und die RCCs HAS_ORDER-MC und HAS_ORDER-OC, wobei Oid (U) Primärschlüssel und Cid (NU) Fremdschlüssel in ORDER sind. Mit CUST.CType oder CUST.Cid als Einstiegspunkte können jetzt Verbundanfragen wie $(\text{C.CType} = x \text{ and } \text{C.Cid} = \text{O.Cid})$ oder $(\text{C.Cid} = y \text{ and } \text{C.Cid} = \text{O.Cid})$ beantwortet werden, sobald x oder y im Cache gefunden werden. Der RCC HAS_ORDER-OC vom Typ n:1 ist in dieser Situation wegen der Symmetrie der wertbasierten Beziehung C.Cid und O.Cid (was der Primär-/Fremdschlüsselbeziehung in den BE-Tabellen CUST und ORDER entspricht) automatisch immer erfüllt[11]; er müsste also nicht explizit spezifiziert werden. O.Oid ist wegen U explizit bereichsvollständig, O.Cid implizit, da ORDER nur über den RCC HAS_ORDER-MC befüllt wird. Deswegen können diese beiden Spalten wiederum als Einstiegspunkte benutzt werden. Das bedeutet, dass zusätzlich Anfragen wie $\text{O.Oid} = v$ oder $\text{O.Cid} = w$ bzw. $(\text{O.Oid} = v \text{ and } \text{O.Cid} = \text{C.Cid})$ oder $(\text{O.Cid} = w \text{ and } \text{O.Cid} = \text{C.Cid})$ im Cache (CG2) beantwortet werden können, sobald der aktuelle Wert v oder w durch Probing im Cache festgestellt wird.

Eine komplexere Cache-Group CG3 lässt sich aus CG2 durch Einbinden von FE-Tabelle ORDER_LINE (kurz OL) und den RCCs HAS-MC und HAS-OC bilden. OLid sei hier Primärschlüssel (U) und Oid (NU) und Iid (NU) seien Fremdschlüssel der entsprechenden BE-Tabellen. Aus den gleichen Gründen wie in CG2 ist OL.OLid explizit und OL.Oid implizit bereichsvollständig. Wie in CG2 können alle bereichsvollständigen Spalten als Einstiegspunkte dienen und jeder RCC kann in Pfeilrichtung als Verbundprädikat genutzt werden.

Noch komplexere Cache-Groups CG4 und CG5 lassen

11. Diese Eigenschaft des Relationenmodells kann hier als eine Art Optimierung für Verbunde genutzt werden, wenn ein geeigneter Einstiegspunkt dafür gefunden wird. Bei referenzbasierten Datenmodellen, bei denen Cache-Referenzen z. B. durch Zeiger repräsentiert werden, würde diese Symmetrieeigenschaft nicht zutreffen.

sich für unser Anwendungsbeispiel durch Einbeziehung von ITEM und IS_ORDERED-OC bzw. AUTHOR und WRITES-OC spezifizieren. Unsere Beobachtungen aus CG3 können wir dabei voll übertragen. Das bedeutet, dass wir beispielsweise einen Verbund von CUST über ORDER, ORDER_LINE, ITEM bis zu AUTHOR im Cache auswerten können, wenn die entsprechenden Gleichheitsprädikate (Einstiegspunkte) dies ermöglichen.

Cache-Groups CG1 bis CG3 sind „angenehm“ zu verwalten, da sie auf der Instanzebene aus nicht überlappenden Bäumen bestehen, wenn man die Ausprägungen der RCCs in Richtung 1:n als Kanten auffasst, die Teilbäume miteinander verbinden. Die Einlagerung und Ersetzung solcher baumartig verknüpfter Satzmengen ist konzeptionell einfach. Diese Aussage gilt für CG4 und CG5 nur noch beschränkt, da wegen der n:1-Beziehung von IS_ORDERED-OC und WRITES-OC mehrere Ausprägungen von ORDER_LINE denselben Vater in ITEM oder AUTHOR besitzen können; es entstehen hier also i. Allg. azyklische Graphen. Beim Einfügen dürfen deshalb keine Duplikate erzeugt werden, wohingegen bei der Ersetzung darauf zu achten ist, dass mit dem letzten Kind in ORDER_LINE auch die zugehörigen Väter in ITEM bzw. AUTHOR entfernt werden.

Als wichtiger Aspekt der Leistungsanalyse von Cache-Groups muss ihr Füllverhalten beachtet und kontrolliert werden. Dieses lässt sich analytisch annähern, wenn wir die vereinfachenden Standardannahmen bei der Anfrageoptimierung unterstellen: Gleichverteilung aller Werte in einer Spalte und stochastische Unabhängigkeit der Werte verschiedener Spalten. Diese Frage soll hier nicht erschöpfend behandelt werden. Überblickshalber lässt sie sich jedoch im einfachsten Fall durch wenige Parameter, auf unser Beispiel in Abb. 14 bezogen, skizzieren. Mit den angegebenen Kardinalitäten für die RCCs vom Typ MC (β, γ) und der Annahme, dass ein Wert von CType α Sätze in CUST, jeder Satz von CUST β Sätze von ORDER und jeder Satz von ORDER wiederum γ Sätze von ORDER_LINE einlagert, lädt jeder Cache-Key-Wert $\alpha(1+\beta(1+\gamma))$ Sätze nach CG3. Bei CG5 kommen noch n Sätze hinzu, wobei diese Größe sich nur mit $2 \le n \le 2(\alpha(1+\beta(1+\gamma)))$ abschätzen lässt. Hier wird die dominierende Rolle der Selektivität der Cache-Key-Spalten vom Typ NU überdeutlich.

Die RCCs HAS_ORDER-OC und HAS-OC verändern das Füllverhalten aller bisher mit Hilfe von Abb. 14 diskutierten Cache-Groups nicht, da weder ORDER noch ORDER_LINE außer über HAS_ORDER-MC und HAS-MC gefüllt werden. Sie sind sozusagen zur expliziten Illustration aller möglicher Verbundrichtungen eingetragen. Das ändert sich aber dramatisch, wenn die RCCs IS_ORDERED-MC und, in schwächerer Weise, WRITES-MC spezifiziert werden würden (beachte: MC, nicht OC). Dann würden nämlich heterogene Zyklen zwischen ITEM, ORDER_LINE, ORDER bzw. sogar CUST entstehen, die in unkontrollierbarer Weise CG4 bzw. CG5 rekursiv füllen würden.

Die zuletzt erörterten Aspekte – quantitatives Füllverhalten und übermäßiges, unkontrollierbares Laden bei heterogenen RCC-Zyklen – zeigen noch einmal eindrucksvoll, dass die Nutzung von Cache-Groups sorgfältig geplant werden muss. Da die Zusammenhänge alles andere als leicht durchschaubar sind, ist es wichtig, ihren Einsatz gut vorzubereiten und durch Werkzeuge zu unterstützen. Dazu sind Fragen zu klären, welche Arten von Cache-Groups sinnvoll, d. h. kosteneffektiv und ohne üble „Performance-Überraschungen“ einsetzbar sind und welcher Gewinn dabei für eine angenommene Anfragelast zu erwarten ist. Die Antworten darauf könnten durch einen Cache-Advisor gegeben werden, nach Möglichkeit weitgehend automatisch.

3.4 Cache-Group-Federations

Wie schon in Abschnitt 3.3.1 skizziert, entwerfen wir als Cache-Administratoren eine konkrete Cache-Group (mit einem einzigen Cache-Key), um damit einen bestimmten Anwendungsfall im Cache zu unterstützen. Zur gleichzeitigen Unterstützung mehrerer Anwendungsfälle sind daher verschiedene Cache-Groups zu überlagern, d. h. die Vereinigung der Graphen aus Tabellen (einige davon mit Cache-Keys) und RCCs zu bilden. Diese Überlagerung mehrerer Cache-Groups bezeichnen wir als *Cache-Group-Federation*. Würde man gleichnamige Tabellen aus mehreren Cache-Groups nicht zu einer Tabelle im FE zusammenfassen, wäre die Anwendungstransparenz gefährdet und müsste ggf. durch aufwändige Replikations- bzw. Verteilungslogik wiederhergestellt werden: Der Benutzer darf nichts von der Existenz verschiedener Instanzen einer Tabelle merken; er gibt einfach den Namen aus dem BE-Schema an.

Bei der Föderation verschiedener Cache-Groups können ihre Cache-Keys und RCCs in Wechselwirkung miteinander treten. Dies kann positive, aber auch negative Effekte

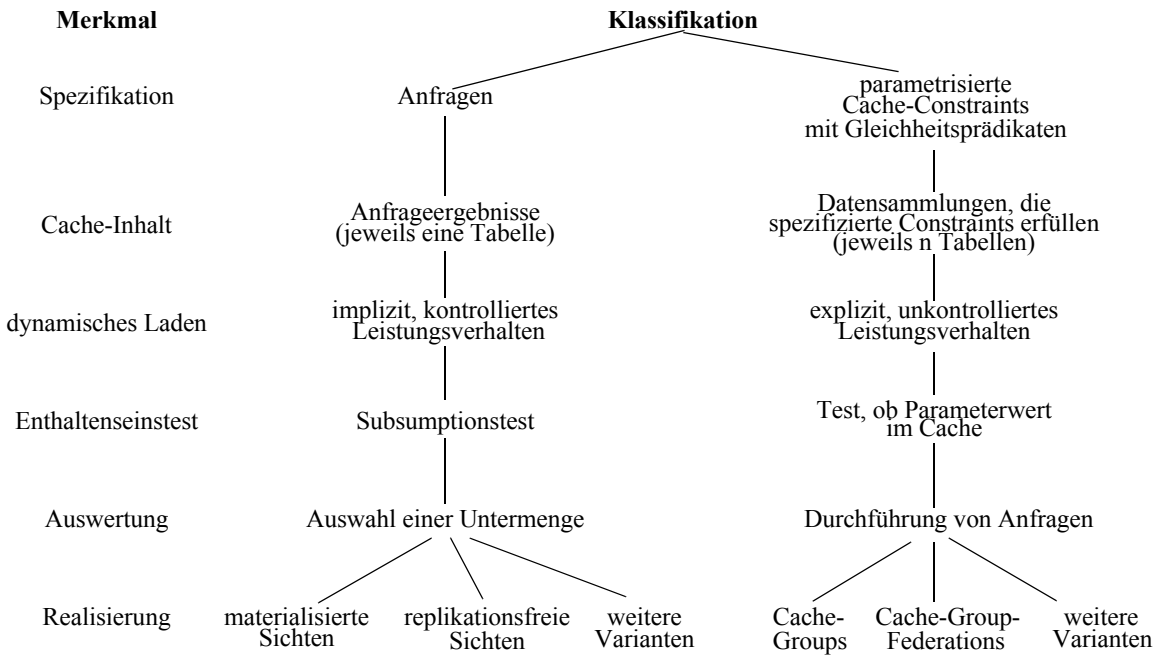


Abb. 15: Klassifikation existierender Ansätze

haben: Im Extremfall entstehen die in Abschnitt 3.3.2 diskutierten Situationen, die rekursives Laden ermöglichen. Aber auch in ungefährlichen Fällen kann durch das Zusammentreffen verschiedener RCC-Netze die Referenz eines Cache-Keys (umfangreiche) Nachladeoperationen in durch die Föderation erreichbaren Tabellen, die aber zu anderen Cache-Groups gehören, auslösen, welche beim isolierten Cache-Group-Entwurf nicht geplant und entsprechend für den Anwendungsfall als unnütz (oder gar unsinnig) erachtet wurden. Positiv wirkt sich ggf. aus, dass Cache-Groups Sätze einer FE-Tabelle gemeinsam benutzen können.

Existieren beispielsweise zwei Cache-Groups, vereinfacht dargestellt durch $A \rightarrow C \rightarrow D$ und $B \rightarrow C \rightarrow E$ mit A und B als Wurzeltabellen, so ist in der entstehenden Cache-Group-Federation die FE-Tabelle C nur einmal vorhanden. Sie wird also über $A \rightarrow C$ und $B \rightarrow C$ geladen, was wiederum bewirkt, dass die vereinigten Satzmengen in C ursprünglich unbeabsichtigte Ladevorgänge in D und E auslösen. Ein positiver Effekt (Gewinn) oder eine Entlastung entsteht in C im Vergleich zur Belegung in den einzelnen beteiligten Cache-Groups, wenn sich die vereinigten Satzmengen überlappen, da keine Duplikate gespeichert werden. Im Allgemeinen übersteigt die Belastung, die eine Cache-Group-Federation den beteiligten Cache-Groups aufbürdet, den zu erwartenden Gewinn bei weitem. Dieser Trade-off kann in praktischen Fällen eine Redefinition, d. h. meist eine Vereinfachung, der beteiligten Cache-Groups erzwingen. Genauere Aussagen dazu erfordern eine Leistungsmodellierung und -analyse, eine mögliche Aufgabe des angesprochenen Cache-Advisors.

4 Ableitung einer Klassifikation

Nachdem wir das Spektrum existierender Lösungen für das DB-Caching analysiert und in verschiedenen Aspekten bewertet haben, ist es für das weitere Durchdringen des Problems hilfreich, eine Klassifikation abzuleiten, um das mögliche Lösungsspektrum besser erkennen und ausloten zu können. In der Literatur [2, 3, 12, 17] wurden bisher immer nur einzelne Implementierungen vorgestellt, die den Blick auf die zugrunde liegenden Ideen und Prinzipien nicht erleichtern. Um existierende Lösungen vergleichen oder neue entwickeln zu können, müssen jedoch die Konzepte hinter den Implementierungen verstanden sein. Deshalb haben wir die existierenden Lösungen in Abb. 15 nach ihren wesentlichen Merkmalen geordnet und klassifiziert

4.1 Anfrage-basierte Verfahren

Anfrage-basierte Verfahren sind konzeptionell einfach und seit langem in der Literatur bekannt [3, 5, 13]. Oft wurden sie jedoch nur im Zusammenhang mit deklarativem Cach-

ing vorgeschlagen. Auch in ihren dynamischen Varianten bleiben sie konzeptionell übersichtlich. Das dynamische Laden der Ergebnismenge geschieht implizit zusammen mit der betreffenden Anfrageauswertung im BE-DBS. Da das Anfrageergebnis ohnehin über das FE-DBS an den Benutzer weitergeleitet wird, führt seine Einlagerung in den Cache nur zu einem geringen Zusatzaufwand. Bei der Realisierung sind weitere Varianten denkbar, die hier nicht entwickelt wurden. Solange Sichten nur eine BE-Tabelle betreffen, können sie replikationsfrei in einer gemeinsamen FE-Tabelle vereint werden, wobei verschiedene Vorgehensweisen bei deren Speicherung und Konsistenzwartung denkbar sind. Jedoch provoziert Caching von Verbundsichten notwendigerweise das Problem, dass bei inhaltlich überlappenden FE-Tabellen Replikation auftritt. Anfrage-basierte Verfahren führen vor der Verwendung des Cache-Inhalts einen direkten Subsumptionstest zwischen den schon gespeicherten Anfrageprädikaten sowie dem zu testenden durch; bei Erfolg kann als Anfrageergebnis eine Untermenge des Cache-Inhalts ausgewählt werden.

4.2 Constraint-basierte Verfahren

In vereinfachter Form wurden Constraint-basierte Verfahren bei Data-Shipping-Architekturen objektorientierter DBMS oder Client/Server-DBMS vorgeschlagen. Dabei ging es um möglicherweise überlappende Constraint-Formeln, die zur Inhaltsbeschreibung eines Client-Caches herangezogen wurden. Jedoch beziehen sich [5, 11] auf die Unterstützung von Anfragen, die sich auf einfache Selektionen beschränken.

Im Vergleich zu den Anfrage-basierten Verfahren sind solche, die auf der Nutzung von parametrisierten Cache-Constraints beruhen, wissenschaftlich gesehen neu und deshalb wenig untersucht. Sie verkörpern, wie die Diskussion in den Abschnitten 3.3 und 3.4 gezeigt hat, auch eine gewisse konzeptionelle Komplexität. Da nach Referenzieren eines sich nicht im Cache befindenden Cache-Key-Wertes oft große Anteile der zugehörigen Prädikatsextension außerhalb der auslösenden Benutzeranfrage dynamisch zu laden sind, kann die zusätzliche Belastung für BE- und FE-DBS erheblich sein. Insgesamt gesehen lässt sich deshalb das Leistungsverhalten nur sehr schwer kontrollieren, obwohl durch eine Reihe von Einschränkungen schon übermäßige und nicht abschätzbare, d. h. rekursive Ladevorgänge verhindert werden. Essentiell für die Verfahrensklasse ist der einfach zu gestaltende Test, ob ein Parameterwert im Cache vorliegt. Bei der Realisierung sind auch hier neben den eingeführten Verfahren weitere Varianten denkbar, die im Einzelnen nicht weiter beleuchtet werden können.

Da Anfrage-basierte Cache-Verfahren Anfragen immer nur aus Anfrageergebnissen (einzelne Tabellen) beantworten können, sind Constraint-basierte Verfahren deutlich flexibler, da sie die Anfragen neu auswerten und vor allem die Ausgaben neu zusammenstellen können.

Die Aspekte der Aktualisierung von Prädikatsextensionen im Cache bzw. deren Ersetzung oder Invalidierung haben wir nicht in die Klassifikation aufgenommen, da alle Verfahren mit allen Caching-Ansätzen kombinierbar sind und ihr Aufwand wesentlich von der Komplexität der verwalteten Prädikatsextensionen abhängt. Schließlich muss das BE-DBS in allen Fällen erkennen, welche geänderten Satzmengen an das einzelne FE-DBS zu schicken sind. Andererseits muss der Cache die (überlappenden) Satzmengen identifizieren, die zu aktualisieren oder zu invalidieren sind.

4.3 Gibt es noch völlig neue Lösungsideen?

Existieren im Kontext unseres Klassifikationsschemas noch andere mögliche Verfahren, die für DB-Caching relevant werden können? Wie in Abschnitt 2.2 gezeigt wurde, ist die Gewährleistung der Vollständigkeitsbedingung in der FE-DB zentral für das DB-Caching. Bei den Anfrage-basierten Verfahren verkörpert das Anfrageergebnis diese direkt. Bei den parametrisierten Cache-Constraints, die wir bisher betrachtet haben, wurde sie dagegen dadurch garantiert, dass alle Werte in den Cache-Key- und Unique-Spalten bereichsvollständig waren und die zugehörigen Sätze die Anwesenheit der über definierte RCCs abhängigen Sätze erzwangen. Bereichsvollständigkeit (als einfachste Form der Prädikatsvollständigkeit) war in einfacher Weise zu testen, und im Erfolgsfall ließen sich Verbunde über die durch RCCs erreichbaren Satzmengen abwickeln. Daraus resultierten dann die für praktische Anwendungsfälle besonders wichtigen Anfrageklassen: Selektionen über Gleichheitsprädikate (siehe CG1) und Verbunde (siehe CG2–CG5). Eine Verallgemeinerung dieser Sichtweise führt uns zu der *Schlüsselbeobachtung*, dass sich auch andere Typen von SQL-Prädikaten im DB-Cache auswerten lassen, wenn sich deren *Prädikatsvollständigkeit* zum Anfragezeitpunkt im Cache feststellen lässt. Auf diese Weise könnte der obige Ansatz erweitert werden für

- einfache Prädikate mit Vergleichsbedingungen $\Theta \in \{<, >, \leq, \neq, \geq\}$
- Bereichsprädikate oder gar mit diesen durch aussagenlogische Operatoren ($\vee$, $\wedge$, $\neg$)
- zusammengesetzte Prädikate.

Weiterhin ist es denkbar, jedoch viel komplexer, Prädikatsvollständigkeit[12] für

- Aggregations-
- Rekursions- und weitere
- SQL-Prädikate (Exists, Subquery usw.)

herzustellen. Deren Nützlichkeit und Realisierungsdetails bzw. -einschränkungen wären natürlich noch genauer zu evaluieren.

Offensichtlich ist die Idee, Prädikatsvollständigkeit im DB-Cache zu bewerkstelligen, nicht auf SQL-Prädikate beschränkt. In gleicher Weise ist es möglich, diese für XML-Daten und XPath- oder XQuery-Prädikate sowie für Objekte anderer Datenmodelle herzustellen.

All diese Erweiterungen, die auf bisher nicht unterstützte Prädikate bei der Anfrageauswertung im Cache abzielen, führen neue Unterklassen Constraint-basierter Verfahren ein, die wir in Abb. 15 nicht explizit aufgenommen haben, um den Teilbaum mit „parametrisierte Cache-Constraints“ als Wurzel nicht zu überlasten. Die Beantwortung der Frage, ob es völlig neue und praktikable Caching-Konzepte, d. h. weitere Teilbäume unter „Klassifikation“ gibt, überlassen wir künftigen Forschungsarbeiten.

5 Zusammenfassung und Ausblick

In diesem Aufsatz haben wir Datenbank-Caching, das vor allem DB-basierte Web-Applikationen hinsichtlich Skalierbarkeit, Leistungsverhalten und Verfügbarkeit verbessern soll, als neue Technik für die DB-Middleware vorgestellt. Dabei haben wir eine Systematik zur Charakterisierung existierender und möglicher Verfahren aufgestellt, die insbesondere auch ihren Vergleich und ihre Bewertung erleichtern soll.

12. Im Einzelfall wäre natürlich die Testkomplexität dieser Prädikate noch zu überprüfen. Im BE-DBS sind diese Tests erforderlich, um die geänderten Satzmengen zu bestimmen, die zur Wartung an den Cache zu schicken sind. Im FE-DBS fallen sie vor allem für Testanfragen und für die Kontrolle überlappender Prädikatsextensionen an.

Zentral für das DB-Caching ist die Gewährleistung der Vollständigkeitsbedingung in der FE-DB hinsichtlich des lokal auszuwertenden Anfrageprädikats. Die Verfahrensklasse der materialisierten Sichten bietet hierfür konzeptionell einfache Lösungsmöglichkeiten, die sich durch die Nutzung replikationsfreier Sichten optimieren lassen. Allerdings erlaubt diese Vorgehensweise immer nur die Auswahl von Teilmengen aus solchen materialisierten Sichten, wenn diese den Subsumptionstest für das aktuelle Anfrageprädikat bestehen.

Eine neue Verfahrensklasse, bei welcher der Cache-Inhalt durch parametrisierte Cache-Constraints bestimmt wird, erlaubt dagegen eine flexible Anfrageverarbeitung im Cache. Im Allgemeinen lassen sich dadurch Anfragen mit deutlich komplexeren Prädikaten lokal in der FE-DB abwickeln. Das bekannte Verfahren der Cache-Groups erfordert Bereichsvollständigkeit von Cache-Keys. Dieses Konzept bildet zusammen mit den Referentiellen Cache-Constraints die Grundlage, Selektionen mit Gleichheitsprädikaten und abhängige Verbundoperationen im Cache durchzuführen.

Die Untersuchung dieser Klasse führte zum Begriff der Prädikatsvollständigkeit, dessen Gewährleistung in verschiedenen Ausprägungen die Auswertung wesentlich komplexerer Anfragetypen im Cache ermöglichen würde. Allerdings ist bei solchen Überlegungen zu berücksichtigen, dass dafür entwickelte Verfahren oft nur bei sehr dedizierten Anfrageprofilen sinnvoll und kosteneffektiv einsetzbar wären. Andererseits wirft Prädikatsvollständigkeit bei komplexen Prädikaten Entscheidbarkeitsprobleme auf, da stets das vollständige Einlagern bzw. Ersetzen der zugehörigen Extensionen (Satzmengen) bewerkstelligt werden muss und bei einer aktuellen Anfrage zu entscheiden ist, ob ihre Auswertung im Cache sicher ist.

Die kurze Betrachtung der Cache-Group-Federations hat verdeutlicht, dass die Spezifikation zu vieler Constraints es sehr schnell erzwingen würde, die ganze DB bzw. den vollständigen, von der Anfragelast betroffenen DB-Ausschnitt im Cache vorzuhalten. Deshalb würde in solchen Situationen kein Weg daran vorbeiführen, häufig frequentierte BE-Tabellen vollständig in die FE-DB zu laden, wenn Anfragen dort abgewickelt werden sollen (Full-Table Caching).

Nach der bisherigen Modellvorstellung finden alle Aktualisierungsoperationen in der BE-DB statt. Nach erfolgreichem Commit müssen dann alle betroffenen Prädikate mit ihren Extensionen in den verschiedenen FE-DBs, über Sub-

skriptionslisten vom BE-DBS kontrolliert, aktualisiert werden, um die Datenkonsistenz zum BE-DB-Zustand wiederherzustellen. Das kann bei Constraint-basierten Verfahren wiederum Nachladevorgänge oder zusätzliche Invalidierungen im Cache auslösen. Denkbar wären aber auch Aktualisierungsmodelle, die alle transaktionalen Änderungen zunächst nur in der FE-DB abwickeln und diese nach Commit zum BE-DBS weiterleiten, das anschließend die anderen betroffenen FE-DBs mit den geänderten Daten versorgt. Andere Modelle könnten darauf abzielen, transaktionale Änderungen gleichzeitig in der aktuellen FE-DB und der BE-DB (unter einem verteilten 2PC-Protokoll) durchzuführen, bevor die Konsistenz in den anderen betroffenen FE-DBs nachgezogen wird. Kostenaspekte wie auch Arten der Datenkonsistenz, die unter den verschiedenen Aktualisierungsmodellen gewährleistet werden können, sind momentan völlig unklar, so dass die Lösung dieser Fragestellungen noch weitere substanzielle Forschungsarbeiten erfordert.

Danksagung.

Wir möchten Mehmet Altinel, Christof Bornhövd und C. Mohan für die zahllosen Diskussionen danken, die der erste Autor 2003 während seines Forschungsaufenthaltes am IBM Almaden Research Center in San Jose führen konnte. Sie haben wesentlich zur Begriffsbildung und zum vertieften Verständnis der DB-Caching-Verfahren beigetragen, die auf der Nutzung von parametrisierten Cache-Constraints beruhen. Weiterhin bedanken wir uns bei Christian Bayerlein, Bernhard Mitschang, Joachim Thomas und den anonymen Gutachtern für hilfreiche Hinweise, die zu einer verbesserten Lesbarkeit dieses Aufsatzes führten.

6 Literatur

[1] Akamai Technologies Inc. Akamai EdgeSuite. http://www.akamai.com/html/en/tc/core.tech.html

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

[3] Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan: DBProxy: A Dynamic Data Cache for Web Applications. Proc. Int. Conf. on Data Engineering 2003, Bangalore, India, S. 821-831

[4] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, Mohamed Ziauddin: Materialized Views in Oracle. Proc. 24th Int. Conf. on Very Large Data Bases, New York, 1998, S. 659-664

[5] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, Michael Tan: Semantic Data Caching and Replacement. Proc. 22th Int. Conf. on Very Large Data Bases, Bombay, 1996, S. 330-341

[6] Jonathan Goldstein, Per-Åke Larson: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution.Proc. Int. Conf. on Management of Data 2001, Santa Barbara, CA, S. 331-342

[7] Theo Härder, Bernhard Mitschang, Udo Nink, Norbert Ritter: Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen, Informatik - Forschung und Entwicklung 10:2, Springer-Verlag, 1995, S. 55-72.

[8] Theo Härder, Erhard Rahm: Datenbanksysteme – Konzepte und Techniken der Implementierung. 2. Auflage, Springer (2001)

[9] IBM DB2 Universal Database (V 8.1), http://www-306.ibm.com/software/data/db2/

[10] IBM Websphere Edge Server. http://www-4.ibm.com/software/webservers/edgeserver/

[11] Arthur Keller, Julie Basu: A Predicate-Based Caching Scheme for Client-Server Database Architectures, VLDB Journal 5:1, 35-47 (1996)

[12] Per-Åke Larson, Jonathan Goldstein, Jingren Zhou: MTCache: Mid-Tier Database Caching in SQL Server. Proc. Int. Conf. on Data Engineering 2004, Boston, MA

[13] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, Divesh Srivastava: Answering Queries Using Views. Proc. Symposium on Principles of Database Systems 1995, San Jose, CA, S. 95-104

[14] Udo Nink: Anbindung von Entwurfsdatenbanken an objektorientierte Programmiersprachen. Shaker-Verlag, 1999

[15] Oracle Internet Application Sever Documentation Library, http:/7technet.oracle.com/docs/products/ias/doc_index.htm

[16] Stefan Podlipinig, Laszlo Böszörmenyi: A Survey of Web Cache Replacement Strategies. ACM Computing Surveys 35:4, 374-398 (2003)

[17] The TimesTen Team: Mid-tier Caching: The TimesTen Approach. Proc. Int. Conf. on Management of Data 2002, Madison, WI, S. 588-593

[18] Gerhard Weikum: Web Caching. In: Web & Datenbanken – Konzepte, Architekturen, Anwendungen. Erhard Rahm/Gottfried Vossen (Hrsg.), dpunkt.verlag, 191-216 (2002)