Einen Schritt zurück zum negativen Datenbank-Caching

Andreas Bühmann

Fachbereich Informatik
Technische Universität Kaiserslautern
Gottlieb-Daimler-Straße
D-67663 Kaiserslautern
buehmann@informatik.uni-kl.de

Zusammenfassung: Ein Schlüssel zur Erhöhung der Qualität von Web-Anwendungen ist Caching. Während das Web-Caching Dokumentfragmente bereithält, die zunehmend aus Datenbank-Daten generiert werden, richtet sich das Datenbank-Caching auf die redundante Speicherung dieser Daten selbst. Eine adaptiv verwaltete Teilmenge der Backend-Daten ermöglicht im Cache durch Vollständigkeitseigenschaften die korrekte Auswertung von Anfragen.

In Cache Groups für Gleichheitsprädikate, einer Ausprägung des Constraint-basierten Datenbank-Caching, kann die Auswertbarkeit einer Anfrage durch einfache Sondierungsanfragen auf dem Cache-Inhalt entschieden werden. Wir stellen ein neues Sondierungsverfahren vor, das den Cache flexibler und für eine größere Anzahl von Anfragen nutzbar macht; dazu gehören unter dem Begriff des negativen Caching auch Anfragen mit leerem Ergebnis.

Wir untersuchen, ob sich das neue Sondierungsverfahren weiter verallgemeinern lässt, welche Alternativen sich für seine Umsetzung in konkreten Cache Groups bieten und wie sich der bisherige Ansatz darin einordnet. Das neue Verfahren macht weiterhin eine Schwäche in der bisherigen Struktur von Cache Groups deutlich, die durch die Einführung von Kontrolltabellen behoben wird, und verschiebt den Schwerpunkt bei der statischen Analyse.

1 Constraint-basiertes Datenbank-Caching

Web-Anwendungen finden heutzutage in den verschiedensten Bereichen ihren Einsatz. Es gibt kaum eine Anwendung, die noch nicht durch Hinzufügen des Präfixes »E-« neu erfunden wurde. Immer mehr solcher (welt-)weit verteilten Anwendungen entstehen; immer mehr weltweit verteilte Benutzer erwarten von diesen Anwendungen den gewohnten Komfort: Kurze Reaktionszeiten, Interaktivität der Bedienung, Aktualität der Information, Zuschnitt auf persönliche Bedürfnisse und ständige Verfügbarkeit des Dienstes.

Auf der anderen Seite haben Anbieter solcher Dienste mit der Erfüllung dieser Anforderungen zu kämpfen: Ein immer größerer Anteil der dem Benutzer präsentierten Information beruht auf Daten, die um ihrer Beherrschbarkeit willen in einem zentralen Datenbanksystem verwaltet werden. Alle Wege von den zahlreichen Benutzern zu den Daten (über Zwischenstationen wie die Anwendungslogik) treffen sich hier, und diese Wege sind lang. Ein allgemeines Mittel zur Verkürzung solcher Wege ist der Einsatz von *Caching*.

Caching bietet immer dann Vorteile, wenn Benutzer mehrfach auf von ihnen benötigte (unveränderte) Daten zugreifen, die sich normalerweise auf einem entfernten Server (*Backend*) befinden. Durch Zwischenspeicherung dieser häufig nachgefragten Daten nahe bei den Benutzern ermöglicht Caching eine Verringerung der Latenzzeit für die Abfrage der Daten, eine Reduktion der Kommunikationskosten, eine Entlastung des Backends sowie eine Steigerung der Skalierbarkeit und Verfügbarkeit des Gesamtsystems.

Der Nutzen von *Web-Caching* [VR02], also dem Caching von statischen oder von der Anwendungslogik generierten Dokumenten oder deren Fragmenten, ist unabstreitbar, aber begrenzt. Zwei Fragmente, die zwar im Wesentlichen auf denselben Daten basieren (etwa eine Liste in zwei unterschiedlichen Sortierungen), gelten für den Web-Cache, der sich irgendwo auf dem Weg zwischen Benutzer und Anwendungslogik befindet, als verschieden.

Das *Datenbank-Caching* konzentriert sich deswegen auf den anderen Teil des Weges, den von der Anwendungslogik zur Datenbank: Ein Datenbank-Cache (*Cache*) enthält eine in letzter Zeit häufig benutzte Teilmenge der Backend-Datenbank, und zwar nicht wie ein Datenbank-Puffer auf der Ebene referenzierter Seiten, sondern in einer Form, die eine Anfragebearbeitung auf dem Cache-Inhalt zulässt [HB04a]. Der Cache sollte auf SQL-Ebene transparent für den Benutzer sein, und unter anderem deshalb bietet es sich an, ihn als Erweiterung eines vollständigen Datenbankverwaltungssystems zu realisieren. (Wir bewegen uns im Folgenden in der Welt der relationalen Datenbanksysteme mit SQL als Anfragesprache, aber die generellen Prinzipien des Datenbank-Caching lassen sich auch auf andere Datenmodelle übertragen.)

Im Gegensatz zu Ansätzen, denen die Wartung recht starr definierter materialisierter Sichten zugrunde liegt [HB04a], strebt das *Constraint-basierte Datenbank-Caching* eine adaptive Verwaltung des Cache-Inhalts an: Dieser soll sich an den in den Vergangenheit nachgefragten Daten orientieren. Wir konzentrieren uns im Folgenden auf die Fragen, wie ein solcher Constraint-basierter Cache beschrieben und mit Daten gefüllt wird und, vor allem, wie wir mit Hilfe des Cache-Inhalts in möglichst flexibler Art und Weise Anfragen beantworten können.

Im verbleibenden Teil von Abschnitt 1 führen wir zunächst die grundlegenden Begriffe zur Beschreibung eines Datenbank-Caches ein, der auf Gleichheitsprädikaten basiert. In Abschnitt 2 zeigen wir, wie man bisher Anfragen in einem solchen Cache bearbeitet hat. Zentral ist dabei das Verfahren zur Entscheidung, ob eine Anfrage überhaupt korrekt beantwortet werden kann, die *Sondierung*. Wir stellen im darauf folgenden Abschnitt 3 eine Verbesserung dieses Sondierungsverfahrens vor, die eine flexiblere Nutzung des Cache-Inhalts erlaubt und das Phänomen des *negativen Caching* ans Licht bringt. In den weiteren Unterabschnitten diskutieren wir die Eigenschaften dieses Verfahrens und mögliche Optimierungen. Schließlich zeigen wir vor einer Zusammenfassung in Abschnitt 4, wie das Verfahren die Cache-Beschreibung zu vereinheitlichen hilft und welche neue Bedeutung es einer statischen Analyse der Cache-Struktur verleiht.

1.1 Cache Groups

Beim Constraint-basierten Datenbank-Caching wird ein Ausschnitt der Backend-Datenbank im Cache abgebildet. Die statische Struktur dieses Ausschnitts wird in Form von

Cache Groups beschrieben.

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

Wir bezeichnen im Folgenden Cache-Tabellen mit Großbuchstaben ($S, T, \ldots$) und ihre Spalten mit Kleinbuchstaben ($e, q_1, \ldots$). Die diesen zugeordneten Backend-Tabellen und -Spalten tragen die gleichen Namen, ergänzt um einen Index B ($S_\mathrm{B}, T_\mathrm{B}, e_\mathrm{B}, q_{1\mathrm{B}}, \ldots$). Wo es auf die Zuordnung von Spalten zu Tabellen ankommt, kann dem Namen einer Spalte *e* der Name der Tabelle *S* vorangestellt werden, in der *e* enthalten ist ($S.e$).

Die Constraints einer Cache Group beschreiben die Inhalte sowie die Abhängigkeiten zwischen den Inhalten der einzelnen Cache-Tabellen und definieren so gültige Zustände des Caches. Die Eigenschaften dieser Zustände können dann herangezogen werden, um zu entscheiden, ob eine gegebene Anfrage aus dem Cache beantwortet werden kann. Constraints werden mit dem Ziel definiert, bestimmte Anfragetypen zu unterstützen und dabei zu gewährleisten, dass die Beantwortbarkeit von Anfragen leicht entschieden werden kann.

Indem der Cache alle definierten Constraints durchsetzt, sorgt er dafür, dass sich die Extensionen bestimmter Prädikate im Cache befinden; er stellt so deren *Prädikatsvollständigkeit* im Cache her [HB04a, HB04d]. Unter der *Extension* eines Prädikats *p* verstehen wir dabei die Menge all derer Sätze, die zur Auswertung von *p* benötigt werden.

1.2 Gleichheitsprädikate

Ein einfacher, aber nützlicher Typ von Cache Groups basiert auf Gleichheitsprädikaten [ABK+03, BAM+04]. Wir betrachten dazu einfache PSJ-Anfragen (nur zusammengesetzt aus Projektion, Selektion und Join). Gleichheitsprädikate in einer PSJ-Anfrage können zum einen eine Spalte $S.e$ auf einen Wert *w* fixieren ($S.e = w$) und zum anderen Equijoins zwischen zwei Spalten $S.a$ und $T.b$ spezifizieren ($S.a = T.b$).

Im Kern sollen Cache Groups, die auf Gleichheitsprädikaten basieren, Anfragen unterstützen, die die Gestalt einer Kette

$$T_1.s_1 = w \wedge T_1.r_1 = T_2.s_2 \wedge T_2.r_2 = T_3.s_3 \wedge \cdots \wedge T_{n-1}.r_{n-1} = T_n.s_n$$

für bestimmte Folgen $s_1, r_1, s_2, \ldots, s_n$ von Spalten und ausgewählte Werte *w* haben. (Wenn wir von einem Prädikat als Anfrage sprechen, meinen wir immer eine PSJ-Anfrage, die sich nur auf die im Prädikat referenzierten Tabellen bezieht; die Projektion ist unwichtig und deswegen beliebig.) Zur Unterstützung solcher Prädikate sind deren Extensionen vollständig im Cache zu halten. Zur Menge der so unterstützten Anfragen kommen in natürlicher Weise weitere Anfragen hinzu, etwa Vereinigungen und Schnitte der Kernanfragen oder Einschränkungen durch zusätzliche per Konjunktion angebundene Prädikate. Zusätzlich zeigt sich später, dass die gewählten Constraints weitere Anfragen unterstützen (auch von der Form der Kernanfragen, aber für weitere Spaltenkombinationen).

Im Rest dieses Abschnittes führen wir die verschiedenen Objekte und Constraints ein, die den skizzierten Anfragetyp unterstützen. Wie sie das im Einzelnen tun, betrachten wir erst in Abschnitt 2.

Zur Unterstützung des Anfangsstücks $T_1.s_1 = w$ der Anfragen und als Mechanismus, um den Cache mit Daten zu füllen, führt man *Füllspalten* ein. Eine Füllspalte ist eine Spalte einer Cache Group, über die das Füllen der Cache Group mit Sätzen aus dem Backend gesteuert wird. Mit einer Füllspalte f (hier $T_1.s_1$) ist eine Menge von Kandidatenwerten, die *Kandidatenmenge*, assoziiert; diese Kandidatenwerte stellen Werte w dar, die bei einer Referenz $f = w$ (enthalten in einer Anfrage, die den Cache erreicht) aus dem Backend in den Cache geladen und dort wertvollständig gemacht werden. (»Laden eines Wertes« bedeutet immer das Laden ganzer Sätze, die diesen Wert enthalten.)

Definition 1. Ein Wert w heißt *wertvollständig* (oder kurz *vollständig*) in einer Spalte $S.e$ genau dann, wenn alle Sätze aus $\sigma_{e=w}S_B$ in S sind.

Im einfachsten Fall umfasst die Kandidatenmenge einer Füllspalte f den gesamten Wertebereich aus der Schemadefinition von f (und damit von f_B). Man kann jedoch zum Beispiel Werte mit einer geringen Selektivität, also Werte, die in einem großen Anteil der Sätze auftreten, aus der Kandidatenmenge ausschließen, damit diese nicht das aufwendige Laden all dieser Sätze in den Cache bewirken.

Um sicherzugehen, dass jeder Wert aus der Kandidatenmenge, der in der Füllspalte im Cache gefunden wird, dort auch vollständig ist, muss die Vollständigkeit auch für Werte hergestellt werden, die auf indirekten Wegen in die Füllspalte gelangen (nicht durch Einfügen nach einer Anfragereferenz, sondern zur Erfüllung anderer Constraints). Die Vollständigkeit von Werten ist wichtig, um Gleichheitsprädikate auf einer Spalte korrekt im Cache auswerten zu können (siehe Abschnitt 2).

In der Regel lassen wir nur eine einzige Füllspalte pro Cache Group zu; wir wollen eine solche Cache Group hier *einfach* nennen. Sollen allerdings mehrere einfache Cache Groups parallel in einem Cache unterstützt werden, ist eine Verschmelzung der verschiedenen Cache Groups nötig, damit jede Backend-Tabelle höchstens einmal im Cache repräsentiert ist. Die so entstehende Struktur, auch als *Cache-Group-Föderation* bezeichnet, hat viele Eigenschaften mit einfachen Cache Groups gemein. Da die Unterscheidung für die folgenden Betrachtungen ohne Belang ist, sprechen wir auch dann (nur) von einer Cache Group, wenn mehrere Füllspalten existieren.

Zur Unterstützung der Equijoins $T_i.r_i = T_{i+1}.s_{i+1}$, also zur Konstruktion von gegenüber $T_1.s_1 = w$ erweiterten Prädikatsextensionen, wird ein Typ von Constraints eingeführt, der eine Beziehung zwischen zwei Cache-Tabellen herstellt: Ein *referentieller Cache-Constraint (RCC)* verknüpft zwei Spalten über eine Werte-Beziehung.

Definition 2. Ein *RCC* $S.a \rightarrow T.b$ von *Quellspalte* $S.a$ zu *Zielspalte* $T.b$ ist genau dann erfüllt, wenn alle Werte w aus $S.a$ vollständig in $T.b$ sind.

Mit anderen Worten sorgt ein RCC also dafür, dass zu Sätzen in S alle Join-Partner aus T_B bezüglich $S.a = T.b$ ebenfalls im Cache sind.

Eine wesentliche Aufgabe der Cache-Verwaltung besteht darin, die Einhaltung der Constraints zu jeder Zeit sicherzustellen. Dazu sind bei Änderungen des Cache-Inhalts, etwa bei Einfügung eines neuen Wertes in eine Füllspalte, Folgeänderungen in den übrigen Tabellen durchzuführen, um beispielsweise die RCCs zu erfüllen. Gleiches gilt für die Entfernung (durch Invalidierung oder Verdrängung) oder Änderung von Sätzen. Der damit verbundenen Problematik (beispielsweise können die von verschiedenen Kandidatenwerten abhängigen Satzmengen überlappen [HB04c]) wollen wir uns hier nicht nähern,

sondern konzentrieren uns auf die Beantwortung von Anfragen aus Cache Groups, die auf Gleichheitsprädikaten basieren.

2 Anfragebearbeitung im Cache

Alle Mechanismen zur Verwaltung des Cache-Inhalts sind ohne Wert, wenn der damit verbundene Aufwand nicht kompensiert wird, indem möglichst viele Anfragen (oder Teile davon) mit Hilfe dieses Inhalts beantwortet werden können.

Beim Constraint-basierten Caching (mit Gleichheitsprädikaten) erfolgt die Anfragebearbeitung in zwei Schritten: Zunächst prüft man, ob eine gegebene Anfrage aus dem Cache beantwortet werden kann (dabei kann auch der beantwortbare Teil der Anfrage bestimmt werden); die eigentliche Planung und Ausführung der Anfrage geschieht dann in herkömmlicher Art und Weise ähnlich wie im Backend. Dies ist möglich, weil der Cache in seinem Schema ausschnittsweise dem Backend entspricht. Im ermittelten Ausführungsplan können dabei sowohl Backend- als auch (an ihrer Stelle) Cache-Tabellen auftreten. Bei der Ausführung des Plans muss in diesem Schritt also nicht darauf geachtet werden, dass die Cache-Tabellen nur einen Teil der Sätze aus den ihnen zugeordneten Backend-Tabellen enthalten: Die Korrektheit der Ersetzung einzelner oder aller Backend- durch Cache-Tabellen und damit die Korrektheit des Anfrageergebnisses wird allein im ersten Schritt sichergestellt.

2.1 Sondierung, Einstiegspunkt und Verankerung

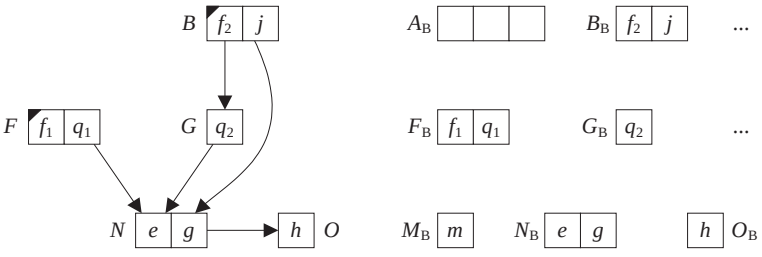
Der erste Schritt der Anfragebearbeitung sollte möglichst effizient durchführbar sein durch einfache Operationen auf dem Cache-Inhalt (Daten und Metadaten). In unserem Fall der Gleichheitsprädikate genügen einfache Testanfragen, die im Cache die Existenz von Werten überprüfen, die in der Anfrage enthalten sind. Diesen Vorgang, bei dem mit Hilfe einfacher Testanfragen die Beantwortbarkeit komplexerer Anfragen entschieden wird, bezeichnen wir als *Sondierung* (englisch *probing*).

Bei Gleichheitsprädikaten des Typs $S.e = w$ genügt ein Test auf die Existenz von w in $S.e$, wenn zusätzlich abgeleitet werden kann (aus der Existenz zusammen mit den Constraints), dass w vollständig im Cache sein muss. Denn dann sind alle Sätze aus S_B, die sich für $S.e = w$ qualifizieren, im Cache. Diese Vollständigkeit muss auf einfache Weise aus dem Cache-Inhalt heraus entschieden werden können. Bei erfolgreichem Abschluss des Sondierens sagen wir, dass die Spalte $S.e$ als *Einstiegspunkt* für das Prädikat $S.e = w$ dient (oder für größere Prädikate, in denen dieses vorkommt).

Ausgehend von einem Einstiegspunkt können ohne weitere Testanfragen Equijoins entlang (und in Richtung) von RCCs korrekt im Cache ausgewertet werden. Existiert also ein RCC $S.a \rightarrow T.b$ und dient $S.e$ bereits als Einstiegspunkt für das Prädikat $S.e = w$, dann kann auch das erweiterte Prädikat $S.e = w \wedge S.a = T.b$ korrekt im Cache ausgewertet werden. Ein Equijoin entlang eines RCCs ist nicht möglich ohne diese Beschränkung in der Quelltabelle des RCCs auf solche Sätze, die zu einer sich im Cache befindenden Prädikatsextension gehören. Bei einem unbeschränkten Join fehlt sonst der Beitrag der nicht im

Cache, aber im Backend vorhandenen Sätze, so dass ein falsches Ergebnis erzeugt wird. Diese Beschränkung als Voraussetzung, einen RCC anwenden zu können, bezeichnen wir auch als *Verankerung* eines RCCs im Cache; entsprechend heißt die Quelltabelle des RCCs in diesem Fall *verankert*.

Die Verankerung von RCCs und Tabellen erfolgt induktiv: Ein Einstiegspunkt in Spalte $S.e$ verankert die Tabelle S und damit alle von S ausgehenden RCCs. Durch die Anwendung eines RCCs $S.a \to T.b$ für einen Equijoin $S.a = T.b$ ist wiederum die Tabelle T verankert, so dass von dort ausgehend weitere RCCs angewandt werden können. RCCs können also stets nur in ununterbrochenen Folgen von Einstiegspunkten ab angewandt werden.



(a) Cache (b) Backend

$B(f_2, j)$	$F(f_1, q_1)$	$G(q_2)$	$N(e, g)$	$O(h)$
$3, \alpha$	$\mathcal{A}, 1$	3	$1, \alpha$	α
$4, \beta$	$\mathcal{B}, 2$	4	$3, \beta$	γ
$7, \beta$	$\mathcal{A}, 3$	[6]	$4, \gamma$	δ
[8, ψ]	[𝒴, 8]	[8]	$6, \alpha$	[ε]
[9, ω]	[𝒵, 9]	[9]	[5, κ]	[ζ]
[0, α]			[5, λ]	[ε]

(c) Belegung

Abbildung 1: (a) Cache-Tabellen repräsentieren eine Auswahl von (b) Backend-Tabellen im Cache und (c) enthalten eine Teilmenge derer Sätze (zusätzliche Sätze im Backend sind eingeklammert).

Beispiel 1. Um zu illustrieren, wie eine Anfrage im Cache behandelt wird, betrachten wir die Konstellation in Abbildung 1; folgende Beispiele beziehen sich immer auf Ausschnitte daraus, wo nötig mit kleinen Änderungen. Die Legende in Abbildung 2 auf der folgenden Seite zeigt die in dieser und allen folgenden Abbildungen verwendeten Symbole.

In der dargestellten Backend-Datenbank (b) existiert eine große Anzahl von Tabellen $A_B, B_B, \ldots, O_B$. Einige wenige daraus (B, F, G, N, O) sind als Cache-Tabellen im Cache widergespiegelt (a). Die Spalten $F.f_1$ und $B.f_2$ sind als Füllspalten deklariert; fünf RCCs stellen die Beziehung zwischen den Cache-Tabellen her: $q_1 \to e$, $f_2 \to q_2$, $q_2 \to e$, $j \to g$ und $g \to h$. Die Belegung des Caches (c) zeigt einen Zustand nach der Referenz der Füllspalten-Werte $f_1 \in \{\mathcal{A}, \mathcal{B}\}$ und $f_2 \in \{3, 4, 7\}$. (Man beginne mit den Sätzen aus F_B und B_B, die diese Werte aufweisen, und stelle dann durch wiederholte Hinzunahme von Sätzen in den übrigen Tabellen die Gültigkeit aller RCCs her.)

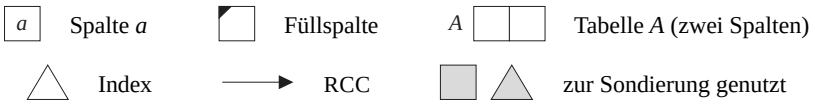


Abbildung 2: Legende

Den Cache erreiche die PSJ-Anfrage eines Clients gegen die Tabellen F, N und M mit dem Prädikat $f_1 = \mathcal{A} \wedge q_1 = e \wedge g = m$. (Wir nehmen an, dass $\mathcal{A}$ zur Kandidatenmenge von f_1 gehört.) Wie stellen wir fest, ob oder zu welchem Anteil diese Anfrage aus dem Cache beantwortet werden kann?

Das Gleichheitsprädikat $f_1 = \mathcal{A}$ bietet eine Möglichkeit zur Verankerung im Cache, da f_1 eine Füllspalte ist und somit alle im Cache vorhandenen Kandidatenwerte dort vollständig sind. Es genügt also, eine Sondierung in f_1 durchzuführen; sie bestätigt, dass sich der Wert $\mathcal{A}$ im Cache befindet. Also kann f_1 als Einstiegspunkt dienen.

Ausgehend von dieser Verankerung garantiert der RCC $q_1 \rightarrow e$ die Korrektheit des Joins $q_1 = e$. Dadurch liegt für Tabelle N eine Verankerung vor, die wir für die Anwendung eines RCCs $g \rightarrow m$ zur Unterstützung von $g = m$ heranziehen *könnten*; aber dieser RCC existiert in unserer Cache Group nicht. Also muss hier auf die Backend-Tabelle M_B zurückgegriffen werden, während wir für F und N die Cache-Tabellen nutzen und somit die Anfrage verteilt zwischen Cache und Backend auswerten können. Natürlich ist diese verteilte Art der Auswertung im Hinblick auf ihre Kosten zu vergleichen mit anderen Alternativen, etwa der vollständigen Ausführung im Backend. Bei der Kostenschätzung sind wesentlich die Kommunikationskosten zu berücksichtigen sowie die Lastsituation im Backend (schließlich soll dieses zu Gunsten der Caches entlastet werden). Darauf gehen wir hier aber nicht weiter ein.

2.2 Optimale Cache-Nutzung

Um den im Cache gehaltenen Datenbestand, der durch die Constraints sowie das Referenzverhalten in der Vergangenheit vorgegeben ist, möglichst effektiv nutzen zu können, muss Klarheit über die verfügbaren Strukturen herrschen, an denen sich die Nutzung orientiert. In unserem Fall ist es wichtig, jeden möglichen Einstiegspunkt zu kennen sowie alle geltenden RCCs zur Unterstützung von Joins.

Bisher wurde in Cache Groups zur einfachen Erkennung und Handhabung von Einstiegspunkten der Begriff der *Bereichsvollständigkeit* einer Spalte eingeführt [ABK+03, HB04b]. Eine Cache-Spalte heißt bereichsvollständig, wenn *jeder* Wert, der in dieser Spalte auftritt, wertvollständig ist. Deswegen genügt beim Sondieren ein bloßer Existenztest in dieser Spalte zur Entscheidung, ob ein Wert vollständig ist und die Spalte somit als Einstiegspunkt dienen kann.

Bei der Analyse von Cache Groups wurde daraufhin die Kenntnis aller bereichsvollständigen Spalten angestrebt. Zum einen wurde in bestimmten Spalten die Bereichsvollständigkeit erzwungen (durch explizite Spezifikation bestimmter Spalten als so genannte *Cache Keys* [ABK+03]; das sind Füllspalten mit maximaler Kandidatenmenge). Zum an-

deren wurden Situationen identifiziert, in denen sich durch das Zusammenspiel von Constraints *induzierte* Bereichsvollständigkeit in Spalten ergibt, für die die Bereichsvollständigkeit nicht explizit spezifiziert worden ist [HB04c, HB04d].

> *Bemerkung.* Ein wichtiger Spezialfall bereichsvollständiger Spalten sind *Unique-Spalten*, also Spalten mit einem Unique-Constraint im Backend. Diese sind unter allen Umständen bereichsvollständig, unabhängig vom Kontext in der Cache Group.

Ähnlich zur induzierten Bereichsvollständigkeit lassen sich auch zusätzliche RCCs finden: Die RCCs, die in einer Cache Group gelten, sind nicht nur die bei der Spezifikation der Cache Group angegebenen RCCs: Im Allgemeinen lässt sich aus diesen die Gültigkeit weiterer ableiten (unter der zusätzlichen Annahme, dass die Cache-Tabellen nur solche Sätze enthalten, die zur Erfüllung der spezifizierten Constraints ausgehend vom Inhalt der Füllspalten benötigt werden, aber nicht mehr). Solche zusätzlichen RCCs wurden von uns *Optimierungs-RCCs* [HB04d] getauft, weil jeder in einer Cache Group bekannte RCC eine neue Möglichkeit für einen Join (in Richtung des RCCs) darstellt und damit die Nutzbarkeit des Caches ohne weitere Kosten optimiert.

Beispiel 2. Ein typischer Optimierungs-RCC in Abbildung 1a wäre $q_2 \rightarrow f_2$: In Spalte q_2 tauchen nur durch den spezifizierten RCC $f_2 \rightarrow q_2$ Werte auf; jeder Wert w, der dort auftaucht, existiert also auch in Spalte f_2. Da die Tabelle B nicht durch andere RCCs erreicht wird, ist jeder Wert w in der Füllspalte f_2 aus der Kandidatenmenge und deswegen vollständig in f_2. Also gilt der entgegengerichtete Optimierungs-RCC $q_2 \rightarrow f_2$.

3 Flexiblere Einstiegspunkte durch ein neues Sondierungsverfahren

Um eine Cache Group möglichst flexibel und effektiv nutzen zu können, galt es bisher, alle geltenden RCCs als mögliche Join-Richtungen zu finden sowie, vor allem, alle bereichsvollständigen Spalten als mögliche Einstiegspunkte, denn erst Einstiegspunkte erlauben die Anwendung von RCCs.

Die Klassifizierung der Spalten der Cache Group in bereichsvollständige und nicht bereichsvollständige, die dabei angestrebt wurde, ist recht grob: Nur Spalten, in denen garantiert (zu jeder Zeit) jeder Wert vollständig ist, werden so als Einstiegspunkte in Erwägung gezogen.

Bei der Suche nach Einstiegspunkten wurde dabei völlig übersehen, dass die Definition von RCCs (Definition 2) auf natürliche Weise ein differenzierteres Sondieren auf Wertvollständigkeit erlaubt: In der Zielspalte eines RCCs sind diejenigen Werte sicher vollständig, die in der Quellspalte des RCCs im Cache vorhanden sind. Deshalb sind in *jeder* Spalte e, die von mindestens einem RCC erreicht wird, (zumindest) alle diejenigen Werte vollständig, die in den Quellspalten eingehender RCCs existieren. Das bedeutet, es reicht aus, in diesen Quellspalten einen Existenztest durchzuführen, um e gegebenenfalls zum Einstieg in die Cache Group nutzen zu können. Es schadet dabei nicht, wenn in der gemeinsamen Zielspalte e weitere Werte existieren, die nicht vollständig sind (in diesem Fall ist e nicht bereichsvollständig).

Beispiel 3. Wir betrachten in Abbildung 3 auf der nächsten Seite einen Ausschnitt aus Abbildung 1a: Die Spalte e wird von zwei RCCs mit Quellspalten q_1 und q_2 erreicht. Soll

die Spalte e als Einstiegspunkt für ein Gleichheitsprädikat $e = w$ dienen, dann genügt eine Sondierung in diesen beiden Spalten, um zu entscheiden, ob der Wert w vollständig in der Spalte e ist. (Existiert w in q_1 oder in q_2, ist w sicher vollständig in e; existiert w weder in q_1 noch in q_2, wissen wir nichts über die Vollständigkeit von w.)

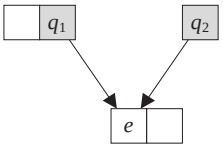


Abbildung 3: Existenztest in den Quellspalten eingehender RCCs

Wir bezeichnen im Folgenden den bisher existierenden und in Abschnitt 2.1 vorgestellten Ansatz, das Sondieren direkt in derjenigen (bereichsvollständigen) Spalte durchzuführen, die als Einstiegspunkt dienen soll, als *altes Sondierungsverfahren*. *Neu* ist das Sondieren in den Quellspalten von RCCs.

Beim Übergang vom alten zum neuen Sondierungsverfahren machen wir, um die Spalten für die Sondierung zu finden, einen Schritt zurück (vom potentiellen Einstiegspunkt entlang der eingehenden RCCs), aber keinen Rückschritt: Das Sondieren in den Quellspalten von RCCs hat mindestens zwei wichtige Vorteile gegenüber dem alten Sondierungsverfahren:

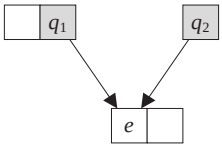
- Erhöhte Flexibilität bei der Benutzung des Caches: Die Bereichsvollständigkeit eines Einstiegspunktes ist nicht mehr notwendig; jede Spalte, die von einem RCC erreicht wird, ist potentieller Einstiegspunkt.
- *Negatives Caching*: Sogar die Information, dass Sätze im Backend nicht existieren, wird im Cache zwischengespeichert und kann zur Beantwortung von Anfragen (mit leerem Ergebnis) genutzt werden. Wir betrachten diese interessante Eigenschaft genauer im folgenden Abschnitt 3.1.

Ein Nachteil ergibt sich in bestimmten Situationen durch einen erhöhten Sondierungsaufwand. In welchen Konstellationen dieser Nachteil zum Tragen kommen, in welchen nicht, und wie sich die Sondierung in Quellspalten optimieren lässt, diskutieren wir in Abschnitt 3.3.

3.1 Negatives Caching

Beispiel 4. Wir betrachten wieder den Ausschnitt aus einer Cache Group in Abbildung 4 auf der folgenden Seite, dieses Mal mit einer konkreten Belegung der Spalten. Spielen wir für verschiedene Anfragen mit Gleichheitsprädikaten auf der Spalte e das neue Sondierungsverfahren durch:

- $e = 4$: Der Wert 4 ist in der Quellspalte q_2 des RCCs $q_2 \rightarrow e$ vorhanden, also ist er vollständig in e: Wir können e als Einstiegspunkt für $e = 4$ nutzen.



Spalte	vorhandene Werte
q_1	$1, 2, 3$
q_2	$3, 4$
e	$\underline{1}, \quad \underline{3}, \underline{4}, \quad 6$
e_B	$1, \quad 3, 4, 5, 6$

Abbildung 4: Negatives Caching: $e = 2$ kann im Cache ausgewertet werden, weil der Wert 2 in der Quellspalte q_1 vorhanden ist, liefert aber ein leeres Ergebnis: Der Wert 2 existiert nicht im Backend (e_B), also auch nicht im Cache (e). (Von unterstrichenen Werten ist bekannt, dass sie vollständig sind.)

- $e = 6$: Der Wert 6 ist in Spalte e vorhanden. Wir nehmen an, dass er auf anderem Wege in diese Spalte gelangt ist als über die dargestellten RCCs. (Wie man Abbildung 1 entnehmen kann, war RCC $j \to g$ dafür verantwortlich.) Der Wert 6 ist aber weder in der Quellspalte q_1 vorhanden noch in q_2. Das bedeutet *nicht*, dass der Wert 6 nicht vollständig in e sein muss; jedoch können wir uns dessen nicht sicher sein. Entsprechend kann e nicht als Einstiegspunkt für $e = 6$ dienen.

- $e = 2$: Der Wert 2 existiert nicht in der Spalte e_B im Backend, dementsprechend auch nicht in der Cache-Spalte e. Allerdings existiert er in einer der Quellspalten (q_1) und ist deswegen in e wertvollständig, *obwohl* er dort nicht auftaucht. Also kann e analog zum Fall $e = 4$ auch als Einstiegspunkt für $e = 2$ genutzt werden. Die Auswertung des Prädikats im Cache liefert dann das korrekte Ergebnis, nämlich ein leeres.

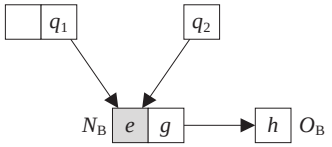
Im Normalfall stehen bei Caching-Verfahren *existierende* Objekte aus dem Backend im Vordergrund; Kopien von ihnen werden im Cache gehalten. Als *negatives Caching* bezeichnen wir den Fall, dass die Information, die zwischengespeichert wird, das Nicht-Vorhandensein von Objekten im Backend ausdrückt. Dieser Begriff taucht bereits beim Caching von DNS-Anfragen auf: “It is the storage of knowledge that something does not exist, cannot or does not give an answer that we call negative caching.” [And98]

In unserem Fall wird durch die Sondierung in Quellspalten von RCCs deutlich, dass eine Cache Group auch Informationen über nicht im Backend vorhandene Sätze enthalten kann. Die Extensionen zu den entsprechenden Gleichheitsprädikaten oder zu darauf aufbauenden Prädikaten sind leer; die leere Antwort auf Anfragen mit diesen Prädikaten kann aber direkt aus dem Cache gegeben werden. Im Gegensatz dazu wird anhand des alten Sondierungsverfahrens in solchen Fällen die Entscheidung getroffen, dass das betroffene Prädikat nicht korrekt im Cache ausgewertet werden kann: Für einen erfolgreichen (korrekten) Einstieg in die Cache Group wird stets die Existenz der angeforderten Werte in der betroffenen Spalte verlangt. Bei Nicht-Existenz muss nach einer Nachricht an das Backend dieses die Bearbeitung übernehmen, nur um anschließend das leere Ergebnis zurückmelden zu können.

Beispiel 5. Wenden wir das alte Sondierungsverfahren auf die Situation aus Beispiel 4 in Abbildung 4 an, finden wir den Wert 2 nicht in Spalte e. Das Prädikat $e = 2$ muss also im Backend ausgewertet werden; dort kommt der Wert 2 in Spalte e_B aber ebenfalls nicht vor.

Jenseits eines erfolgreichen Einstiegs in die Cache Group, bei der Ausnutzung von RCCs für Joins, ist das negative Caching schon im alten Modell des Sondierens und der Cache-Benutzung enthalten, blieb aber bisher unbemerkt.

Beispiel 6. Abbildung 5 zeigt eine Situation im Cache, die wir auf ihr Verhalten bei einer Anfrage mit dem Prädikat $e = 3 \wedge g = h$ hin untersuchen wollen. Dabei legen wir das alte Sondierungsverfahren zu Grunde. Für dieses Beispiel ignorieren wir außerdem den RCC $j \to g$ aus Abbildung 1a. Erst dann kann Spalte e als induziert bereichsvollständig erkannt werden und daher potentieller Einstiegspunkt sein.



Spalten	vorhandene Werte
(e,g)	$(\underline{1},\alpha),(\underline{3},\beta),(\underline{4},\gamma)$
h	$\underline{\alpha}$, $\underline{\gamma}$, δ
h_B	α, $\underline{\gamma}$, δ,ε,ζ

Abbildung 5: Negatives Caching beim alten Sondierungsverfahren: Das Prädikat $e = 3 \wedge g = h$ führt im Cache zu einem leerem Ergebnis.

Die Sondierung in e zeigt, dass der Wert 3 im Cache vorhanden ist, also wegen der Bereichsvollständigkeit dort auch wertvollständig ist. Wegen des RCCs $g \to h$ kann ausgehend von dieser Verankerung in Tabelle N der Join $g = h$ zu Tabelle O korrekt ausgewertet werden. Somit kann die gesamte Anfrage aus dem Cache beantwortet werden.

Der einzige Satz aus N mit $e = 3$ ist $(\underline{3},\beta)$, der g-Wert β findet aber in Spalte h keine Entsprechung. Das Ergebnis der Anfrage ist also leer – ein Fall von negativem Caching.

3.2 Transitivität

Wir sind nun grundlegend vertraut mit dem neuen Verfahren, in Quellspalten zu sondieren: Von der Spalte, die als Einstiegspunkt dienen soll, machen wir einen Schritt zurück entlang der RCCs, um die Spalten für die Sondierung zu finden.

Die Sondierungsspalten können wiederum von RCCs erreicht werden. Bei solchen Ketten von RCCs stellt sich die Frage, ob sich die Idee der Sondierung nicht transitiv fortsetzen lässt: Kann man zur Sondierung auch zwei Schritte zurückgehen? (Dies würde eine flexible Wahl der Sondierungsspalten ermöglichen.)

Wir betrachten eine Kette von RCCs $a \to b \to c$ über drei Spalten a, b und c: Jeder Wert aus a ist vollständig in b; jeder Wert aus b ist vollständig in c. Ist damit auch jeder Wert aus a vollständig in c; gilt der RCC $a \to c$? Ist also die durch eine Menge von RCCs definierte (mathematische) Relation über den Spalten transitiv?

Im Allgemeinen gilt dies nicht. – Nicht ohne Grund stellen wir diese Frage nach der Betrachtung des negativen Caching: Dort wurde uns bewusst, dass ein Wert auch dann vollständig in einer Spalte im Cache sein kann, wenn er dort gar nicht existiert.

Nehmen wir also an, wir haben einen Wert w, der im Backend in a_B und c_B, aber nicht in b_B existiert. Weiterhin sei w im Cache in Spalte a vorhanden. Dann ist w durch den RCC $a \to b$ wertvollständig in b, kann aber dort natürlich nicht existieren. Also macht der

zweite RCC $b \to c$ keinerlei Aussage über die Wertvollständigkeit von w in c, da dies die Existenz von w in b voraussetzt. Sei nun w nicht in c enthalten; dann ist w nicht vollständig in c und unser Gegenbeispiel ist komplett.

Es ist also sorgfältig darauf zu achten, die Existenz von Werten im Cache und deren Vollständigkeit voneinander zu trennen. Dem eben provozierten Irrtum, dieses nicht zu tun, sind Altinel et al. erlegen: Sie beweisen den Satz [ABK+03, Abschnitt 3.4.2], dass jede an einem homogenen Zyklus beteiligte Spalte bereichsvollständig sei. (Ein homogener Zyklus ist eine geschlossene Kette von RCCs im obigen Sinne, z. B. $c_1 \to c_2 \to \cdots \to c_n \to c_1$.) Dieser Beweis ist fehlerhaft; er enthält als wesentliches Element den falschen Schluss, dass ein vollständiger Wert v_j in einer Zyklusspalte c_i über den RCC $c_i \to c_{i+1}$ die Wertvollständigkeit von v_j in der folgenden Spalte c_{i+1} bewirke. Dies (und der ganze Satz) lässt sich durch das obige Gegenbeispiel widerlegen.

3.3 Optimierte Sondierung

Bisher haben wir das neue Sondierungsverfahren in allgemeiner Form dargestellt (für beliebige Spalten, die von mindestens einem, aber sonst beliebig vielen RCCs erreicht werden) und seine Eigenschaften auf dieser Ebene analysiert. Im Folgenden wollen wir betrachten, welche Alternativen es für die Durchführung der Sondierung in konkreten Situationen gibt.

Zuallererst kann man in direkter Art und Weise in Quellspalten sondieren: Soll eine Spalte e als Einstiegspunkt für ein Gleichheitsprädikat $e = w$ dienen, wird jede Quellspalte eines RCCs, der e erreicht, auf die Existenz des Wertes w hin überprüft. Findet man den Wert in einer dieser n_e Spalten ($n_e \geq 0$), ist w vollständig in e, und der Einstieg in die Cache Group ist gesichert. Ein Nachteil dieses Vorgehens ist, dass pro eingehendem RCC ein Existenztest auszuwerten ist, was durch den gesteigerten Aufwand (für $n_e > 1$), der bei jeder Anfrage auf die Spalte e anfällt, die Effektivität des Caching mindert. (Natürlich kann man während des Sondierens sofort abbrechen, wenn zum ersten Mal ein Existenztest erfolgreich abgeschlossen wird. Deswegen könnte man sich hier auch Gedanken machen über eine optimale Sondierungsreihenfolge der eingehenden RCCs.)

Als Optimierung für das Sondieren bietet sich die Verwendung eines speziellen Indexes an, der (bezogen auf eine bestimmte Spalte e) die Vereinigung der Wertemengen aller Quellspalten enthält. Ein solcher *Vollständigkeitsindex* für e enthält damit alle Werte, deren Wertvollständigkeit in e bekannt ist.

> *Bemerkung.* In der Spalte e selbst kann es Werte geben, die vollständig in e sind, ohne dass dieses im Cache festgestellt werden könnte. Dementsprechend können solche Werte auch nie zum Einstieg in eine Cache Group benutzt werden. (Man erinnere sich beispielsweise an den Wert $e = 6$ in Beispiel 4.)

In Abbildung 6a auf der nächsten Seite ist ein Vollständigkeitsindex (als Dreieck; siehe auch Abbildung 2) für die Spalte e in der Cache Group dargestellt, die schon aus Abbildung 4 bekannt ist. Dieser Index ist dabei wie eine Spalte durch RCCs in die Cache Group eingebunden; diese Darstellung wurde gewählt, weil sich ein Vollständigkeitsindex wie eine an dieser Stelle eingefügte separate Spalte i verhält, wenn man für die zugehörige

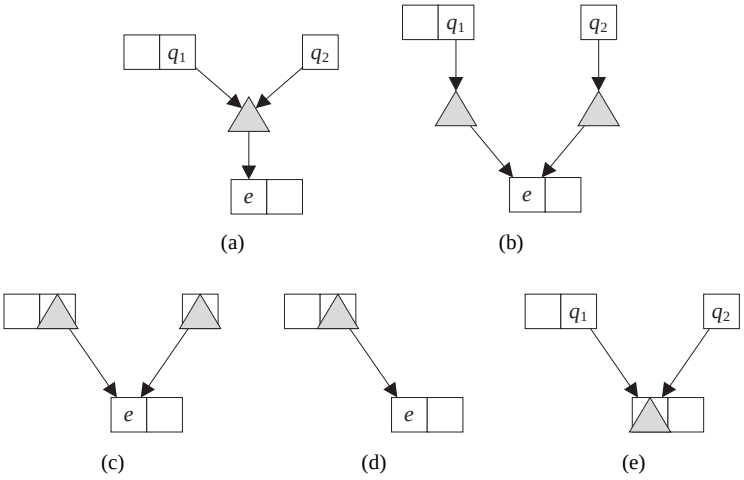


Abbildung 6: Einsatz von Indexen: (a) Sondierung in Vollständigkeitsindex für e anstatt in den Quellspalten q_1 und q_2; (b) naives Sondieren mit normalen Indexen; (c) Abkürzung für (b); (d) nur eine Quellspalte: normaler Index ist Vollständigkeitsindex; (e) bei bereichsvollständigem e (siehe Beispiel 4) Verlust des negativen Caching.

virtuelle (weil nicht existente) Backend-Spalte i_B annimmt, dass sie jeden beliebigen Wert enthält.

Natürlich wird man auch bei Verzicht auf einen Vollständigkeitsindex die Existenztests in den verschiedenen Quellspalten durch einzelne Indexe herkömmlicher Art unterstützen wollen. In Abbildung 6c sind diese Indexe durch Überdeckung der durch sie indexierten Spalten dargestellt; dies kann als Abkürzung der in Abbildung 6b gezeigten Folge von Spalte, RCC und Index aufgefasst werden. Im Falle, dass zu einer Spalte e nur eine Quellspalte q_1 gehört, fallen dieser Index auf q_1 und der Vollständigkeitsindex für e zusammen (Abbildung 6d). In solchen Situationen fallen also keine zusätzlichen Kosten für einen Vollständigkeitsindex an. Wir erwarten, dass diese Situationen – typisch für baumstrukturierte Cache Groups – deutlich häufiger auftreten als solche, in denen eine Spalte von zwei oder mehr RCCs erreicht wird.

Das alte Sondieren direkt in der Spalte e (siehe Abschnitt 2) reiht sich nun in diesem Kontext als eine weitere mögliche Optimierung ein. Sie lässt sich nur dann anwenden, wenn man ableiten kann, dass e stets bereichsvollständig ist. Abbildung 6e zeigt in Anlehnung an Beispiel 4 eine solche Situation. Allerdings verliert man bei dem Schritt von den Quellspalten zur Spalte e einen Teil der Information über vollständige Werte: Die Sondierung kann dann nur noch für solche Werte erfolgreich sein, die in e tatsächlich existieren. Man büßt also das negative Caching ein.

Integritätsbedingungen (*auch* »Constraints«) im Backend sind zu unterscheiden von im Cache definierten Constraints. Ein im Backend gültiger Constraint kann, muss aber nicht im Cache gelten, da sich dort nur eine Teilmenge aller Sätze befindet.

Es gibt Situationen, in denen wegen Integritätsbedingungen, die im Backend definiert sind, nie negatives Caching auftreten kann. Haben wir zwei Spalten f und p mit einem

RCC $f \to p$, und stehen f_B und p_B im Backend in einer Fremdschlüssel–Primärschlüssel-Beziehung, können wegen der referentiellen Integrität nie Werte in f auftauchen, die nicht auch in p existieren. In solchen Situationen bietet es sich daher an, mit dem alten Sondierungsverfahren direkt in der Zielspalte p zu sondieren (p_B ist eine Unique-Spalte, also ist p bereichsvollständig); man verliert dabei keinerlei Information. Allerdings hat man auch in diesem Fall weiterhin die Option, f statt p zur Sondierung heranzuziehen, um etwa einen dort vorhandenen Index auszunutzen.

Es gibt also abhängig von der Zahl der eingehenden RCCs, abhängig von den im Cache definierten Indexen und abhängig von den im Backend definierten Integritätsbedingungen eine Vielzahl von Möglichkeiten, die Sondierung in Quellspalten umzusetzen.

3.4 Füllspalten

Bei den Füllspalten kann das beschriebene Verfahren zum Sondieren nicht ohne weiteres eingesetzt werden; im Allgemeinen müssen diese nicht von RCCs erreicht werden. Der Schritt zurück zum Sondieren ist hier also nicht möglich. Dieser Makel lässt sich für eine Füllspalte f_1 konzeptionell durch die Einführung einer separaten *Kontrolltabelle* mit nur einer Unique-Spalte k und einem RCC $k \to f_1$ von dieser Spalte auf die Füllspalte beheben. Allein die Einträge in dieser Kontrolltabelle bestimmen damit, welche Werte vollständig in die Füllspalte geladen werden (Abbildung 7). Der zulässige Wertebereich von k ist genau die Kandidatenmenge von f_1.

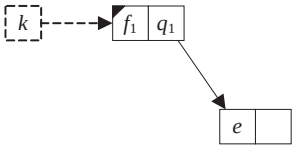


Abbildung 7: Spalte f_1 ist Füllspalte. Eine Kontrolltabelle mit Spalte k und ein RCC $k \to f_1$ führen dazu, dass man zum Einstieg in f_1 bei der Sondierung genau so verfahren kann wie bei jeder anderen Spalte.

Bei genauerer Betrachtung behebt diese Umstrukturierung auch ein weiteres Problem: Bisher wurde keine Trennung vorgenommen zwischen Kandidaten-Werten, die direkt referenziert wurden, und solchen, die auf anderem Wege in die Füllspalte gelangten: Beide wurden in gleicher Weise vollständig gemacht. Die neue Struktur berücksichtigt diesen Unterschied: Nur referenzierte Kandidaten-Werte stehen in der Kontrolltabelle und machen diese Werte in der Füllspalte wertvollständig; andere Werte, die die Füllspalte auf anderen Wegen erreichen, müssen dort nicht unbedingt vollständig erscheinen, selbst wenn es sich um Kandidatenwerte handelt. Das verhindert unnötigen Zusatzaufwand bei der Wartung des Cache-Inhalts, denn jeder unnötig in den Cache geladene Satz kann durch RCCs weitere Sätze mit sich ziehen.

Eine derartige Kontrolltabelle, die über ihren Inhalt die Belegung der eigentlichen Cache-Tabellen kontrolliert, taucht auch beim System MTCache auf [LGGZ04, LGZ04]. Dort werden die Inhalte der Cache-Tabellen als materialisierte Sichten beschrieben, die

sich in ihrer Definition auf die Kontrolltabelle oder andere Cache-Tabellen beziehen können. In der Tat wird durch diesen Bezug (zumindest bei den Sichten im angegebenen Beispiel [LGGZ04, Abschnitt 5]) der Effekt einer Art von RCC beschrieben.

3.5 Bedeutung von Optimierungs-RCCs

Mit der Kenntnis des neuen Sondierungsverfahrens ist die Erkennung von Bereichsvollständigkeit nicht mehr so relevant wie bisher angenommen. Jede Spalte, die von einem RCC erreicht wird, ist potentieller Einstiegspunkt, ohne dass wir irgendetwas über ihre Bereichsvollständigkeit wissen müssen. Unique-Spalten können zusätzliche Einstiegspunkte sein; ihre Bereichsvollständigkeit ist von vornherein immer bekannt. Die Ableitung der Bereichsvollständigkeit einer Spalte ermöglicht damit nur eine optimierte Sondierung in dem Fall, dass eine Spalte mehr als einen eingehenden RCC besitzt und man bereit ist, auf das negative Caching zu verzichten.

Stattdessen gewinnt die Kenntnis und Klassifizierung aller in einer Cache Group geltenden RCCs umso mehr an Bedeutung. Das Problem der Ermittlung von Optimierungs-RCCs stellt sich in zwei Varianten dar:

- Gegeben eine Menge von RCCs (z. B. die Menge der in einer Cache Group definierten), welche RCCs gelten über diese hinaus? Diese Art von Optimierungs-RCCs erschließt erweiterte Join-Möglichkeiten; dies war die bisherige Motivation für die Suche nach Optimierungs-RCCs.

- Wiederum gegeben eine Menge von RCCs, welche RCCs *aus dieser Menge* sind Optimierungs-RCCs bezüglich der übrigen? Welche der RCCs tragen also nicht zum Füllen der Cache Group bei, welche sind in diesem Sinne redundant?

Die zweite Formulierung des Problems verdient besondere Beachtung: Das Erkennen von überflüssigen oder redundanten RCCs führt zu Einsparungen sowohl bei der Wartung des Cache-Inhalts als auch bei der Sondierung.

Redundante RCCs müssen auf der einen Seite nicht beachtet oder überprüft werden, wenn es nach einer Änderung in den Kontrolltabellen (oder darauf folgend in den Cache-Tabellen) darum geht, einen gültigen Cache-Zustand wiederherzustellen.

Auf der anderen Seite können redundante RCCs bei der Bestimmung der Quellspalten eingehender RCCs ausgespart werden, wenn eine Sondierung für ein Gleichheitsprädikat durchgeführt werden soll. Dies verringert die Anzahl derjenigen Spalten, in denen ein Existenztest durchgeführt oder deren Änderungen in einem Vollständigkeitsindex nachgeführt werden müssen, *ohne* Information über vollständige Werte zu verlieren oder die Korrektheit der Auswertung im Cache zu beeinträchtigen.

Um schon beim Entwurf von Cache Groups und daraus entstehenden Cache-Group-Föderationen Klarheit über die Folgen und insbesondere Kosten des Entwurfs zu haben [HB04b], wird also ein Algorithmus (vielleicht auf der Grundlage eines Kalküls) zur Bestimmung aller Optimierungs-RCCs einer Cache Group benötigt.

Bemerkung. Steht erst ein Algorithmus zur Bestimmung aller Optimierungs-RCCs einer Cache Group zur Verfügung, ist damit auch die Ermittlung aller (induziert)

bereichsvollständigen Spalten möglich: Eine bereichsvollständige Spalte $S.e$ kann nämlich charakterisiert werden durch die Existenz eines *Selbst-RCCs* $S.e \rightarrow S.e$.

Beispiel 7. Abbildung 8a zeigt eine Cache Group in der Form, wie sie spezifiziert worden sein könnte, mit einer Füllspalte und fünf explizit angegebenen RCCs. In Abbildung 8b sind zusätzlich die in dieser Cache Group geltenden Optimierungs-RCCs eingezeichnet. Dabei sind sowohl neue RCCs hinzugekommen (z. B. $q_1 \rightarrow e$) als auch spezifizierte RCCs als redundant erkannt worden ($q_1 \rightarrow q_2$). Die ermittelten Selbst-RCCs $q_1 \rightarrow q_1$, $q_2 \rightarrow q_2$ und $f_2 \rightarrow f_2$ zeigen an, dass die Spalten q_1, q_2 und f_2 bereichsvollständig sind.

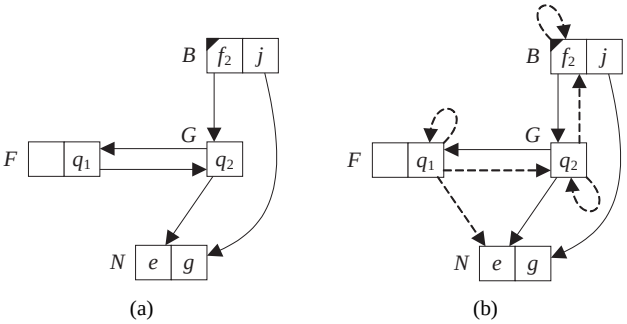


Abbildung 8: Optimierungs-RCCs: (a) Cache Group mit spezifizierten RCCs, (b) mit abgeleiteten und als redundant erkannten RCCs (gestrichelt).

4 Zusammenfassung und Ausblick

Eine effektive Nutzung des Inhalts eines Constraint-basierten Datenbank-Caches beginnt bei der Wahl eines auf die benutzen Cache-Constraints zugeschnittenen und möglichst flexiblen Sondierungsverfahrens. Das alte Sondierungsverfahren für Cache Groups, die auf Gleichheitsprädikaten basieren, stützt sich (zu) stark auf die Bereichsvollständigkeit als Eigenschaft, die für eine ganze Spalte gilt und für *jeden* Wert einer solchen Spalte die Wertvollständigkeit im Cache sichert. Indem es nur bereichsvollständige Spalten als potentielle Einstiegspunkte ansieht, macht das Verfahren sich die Sondierung zwar sehr einfach, ignoriert aber all jene Spalten, die zu irgendeiner Zeit einen einzigen nicht vollständigen Wert enthalten könnten.

Mit dem neuen Sondierungsverfahren machen wir klar, dass es bei der Frage nach dem korrekten Einstieg in eine Cache Group nur auf die Vollständigkeit einzelner Werte ankommt. Anders als im alten Verfahren wird deshalb die Sondierung nicht direkt auf einem potentiellem Einstiegspunkt durchgeführt, sondern auf Spalten, die durch einen Schritt zurück in der Cache-Group-Struktur erreicht werden, nämlich auf den Quellspalten der eingehenden RCCs. Die RCCs selbst sichern zu, dass Werte, die dort gefunden werden, wertvollständig in ihrer Zielspalte sind.

Durch diese Maßnahme ist jede Spalte, die von mindestens einem RCC erreicht wird, potentieller Einstiegspunkt, nicht mehr nur jede bereichsvollständige Spalte. Das führt zu

einer größeren Flexibilität bei der Nutzung des Cache-Inhalts; es können mehr Anfragen aus dem Cache beantwortet werden als vorher. Bei Anfragen, die sich auf Satzmengen beziehen, die leer sind, spielt das durch das neue Sondierungsverfahren aufgedeckte negative Caching seine Vorteile aus: Mit dem im Cache gespeicherten Wissen (aus einer vorangegangenen Anfrage), dass Sätze nicht existieren, kann eine leere Antwort direkt im Cache erzeugt werden, ohne das Backend zu belasten.

Welchen Einfluss negatives Caching im Verhältnis zum üblichen (positiven) Caching auf die Leistungsfähigkeit eines Datenbank-Caches hat, hängt sicherlich von der Art der (Web-)Anwendung ab und der relativen Häufigkeit, mit der diese gezielte Anfragen auf nicht vorhandene Objekte (etwa Benutzer, Produkte oder Bestellungen) durchführt. Die Analyse des Anfrageverhaltens einer konkreten Anwendung über einen Zeitraum könnte hierüber Aufschluss geben.

Die Einführung von Kontrolltabellen in Cache Groups führt dazu, dass alle Spalten in gleicher Art und Weise auf ihre Eignung als Einstiegspunkt überprüft werden können. Verbunden damit ist die strikte Trennung von direkt und indirekt in eine Füllspalte gelangten Kandidatenwerten; nur erstere werden beim Einsatz von Kontrolltabellen wertvollständig gemacht. Das ergibt Sinn, da nur diese Werte direkt aus Anfragen, die den Cache erreichen, stammen.

Beim Einsatz des neuen Sondierungsverfahrens sind je nach Konstellation in mehr Spalten Existenztests durchzuführen als beim alten Verfahren. Eine wichtige Ausnahme ist der häufig anzutreffende Fall, dass es nur einen eingehenden RCC für eine Spalte gibt. In den anderen Fällen können durch übliche Indexierung oder einen speziellen Vollständigkeitsindex die Kosten für die Sondierung klein gehalten werden. Selbst das alte Sondierungsverfahren kann – im Falle geeigneter Backend-Constraints sogar ohne Verlust des negativen Caching – als spezielle Optimierung dienen. Es ergibt sich so auch bei der Durchführung der Sondierung eine große Flexibilität.

Die Kenntnis aller Optimierungs-RCCs einer Cache Group ist nach Einführung des neuen Sondierungsverfahren in zwei Facetten von Bedeutung: Zum einen verbessert sie die Nutzbarkeit des Caches (durch zusätzliche Join-Richtungen), zum anderen minimiert sie den Aufwand bei der Verwaltung des Cache-Inhalts.

Gleichheitsprädikate und die sie unterstützenden Constraints stellen einen sehr einfachen Fall des Constraint-basierten Caching dar. Durch welche Constraints andere Prädikatstypen unterstützt werden können, und wie diese mit den bestehenden wechselwirken, falls sie simultan in einem Cache erfüllt werden sollen, ist eine offene Frage. Schon die Überlagerung mehrerer einfacher (und harmloser) Cache Groups kann durch RCC-Zyklen rekursives und damit unkontrollierbares Füllen des Caches hervorrufen.

Natürlich sind die Anfragebearbeitung im Cache und die dazu nötige Sondierung nur ein Aspekt auf dem Weg zu einem effektiven adaptiven Datenbank-Cache. Weitere Aspekte umfassen die Entwurfsregeln für Cache Groups, die Bewertung der Kosten und des Nutzens, die Ermittlung von Referenzhäufigkeiten oder ähnlichem zur Anwendung von Verdrängungsverfahren, die effiziente Entfernung von gegebenenfalls überlappenden Prädikatsextensionen aus dem Cache sowie verschiedene Strategien zur Bearbeitung und Propagierung von Änderungen im Cache und im Backend. Weiter reichende Fragestellungen richten sich beispielsweise auf die Auflösung der starren Rollenverteilung Cache–Backend

und auf die Möglichkeit, die Struktur einer Cache Group selbst (nicht nur deren Inhalt) an ein verändertes Anfrageprofil anzupassen.

Literatur

[ABK+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh und Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003*, Seiten 718–729. Morgan Kaufmann, 2003.

[And98] M. Andrews. Negative Caching of DNS Queries (DNS NCACHE). Request for Comments (RFC) 2308, März 1998. ftp://ftp.rfc-editor.org/in-notes/rfc2308.txt.

[BAM+04] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh und Berthold Reinwald. Adaptive Database Caching with DBCache. *Data Engineering Bulletin*, 27(2):11–18, Juni 2004.

[HB04a] Theo Härder und Andreas Bühmann. Datenbank-Caching – Eine systematische Analyse möglicher Verfahren. *Informatik – Forschung und Entwicklung*, 19(1):2–16, Juli 2004.

[HB04b] Theo Härder und Andreas Bühmann. Database Caching – Towards a Cost Model for Populating Cache Groups. In *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Lecture Notes in Computer Science* 3255, Seiten 215–229. Springer, 2004.

[HB04c] Theo Härder und Andreas Bühmann. Query Processing in Constraint-Based Database Caches. *Data Engineering Bulletin*, 27(2):3–10, Juni 2004.

[HB04d] Theo Härder und Andreas Bühmann. Value Complete, Domain Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups. http://wwwdvs.informatik.uni-kl.de/pubs/papers/HB04.Magic.html, 2004.

[LGGZ04] Per-Åke Larson, Jonathan Goldstein, Hongfei Guo und Jingren Zhou. MTCache: Mid-Tier Database Caching for SQL Server. *Data Engineering Bulletin*, 27(2):35–40, Juni 2004.

[LGZ04] Per-Åke Larson, Jonathan Goldstein und Jingren Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004*, Seiten 177–189. IEEE Computer Society, 2004.

[VR02] Gottfried Vossen und Erhard Rahm, Hrsg. *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*, Kapitel 7, Seiten 191–216. dpunkt.verlag, Heidelberg, September 2002.

The three most important parts
of any Internet application are caching,
caching, and, of course, caching ...

Larry Ellison, Oracle CEO (2001)