

Andreas Bühmann

Ein Schritt zurück ist kein Rückschritt

Mit flexibler Sondierung zum negativen Datenbank-Caching

the date of receipt and acceptance should be inserted later

Zusammenfassung Ein Schlüssel zur Erhöhung der Qualität von Web-Anwendungen ist Caching. Während Web-Caching Dokumentfragmente bereithält, die zunehmend aus Datenbank-Daten generiert werden, richtet sich 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 präsentieren ein neues Sondierungsverfahren, das den Cache flexibler und für eine größere Anzahl von Anfragetypen 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 außerdem eine Schwäche in der bisherigen Struktur von Cache Groups deutlich, die durch die Einführung von Kontrolltabellen behoben werden kann, und kann zur Wartung des Cache-Inhalts beitragen.

Schlüsselwörter Datenbanksysteme · Datenbank-Caching · Constraint-basiert · Sondierung · Einstiegspunkte · negatives Caching · Kontrolltabellen

Abstract A key to increasing the quality of web applications is caching. While web caching holds document fragments ready, which are increasingly generated from database data, database caching focuses on the redundant storage of these data themselves. By means of completeness properties,

an adaptively managed subset of the backend data permits the correct evaluation of queries in the cache.

In cache groups for equality predicates (that is, in an instance of constraint-based database caching) the evaluability of a query can be determined through simple probe queries on the cache contents. We present a new probing procedure that makes the cache usable for more types of queries in a more flexible way; embraced by the term of negative caching, this even includes queries with empty results.

We investigate whether the new probing procedure can further be generalized, which alternatives exist for its realization in concrete cache groups, and how the previous approach fits into these. Additionally, the new procedure reveals a shortcoming in the current structure of cache groups, which can be remedied by introducing control tables, and aids in maintaining the cache contents.

Keywords database systems · database caching · constraint-based · probing · entry points · negative caching · control tables

CR Subject Classification C.2.4 · H.2.4

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 Information, die dem Benutzer präsentiert wird, beruht auf Daten, die um ihrer Beherrschbarkeit willen in einem zentralen Datenbanksystem verwaltet werden. Alle Wege von den unzähligen Benutzern zu den

Dieser Artikel ist eine bearbeitete und erweiterte Version von [4].

Andreas Bühmann
Fachbereich Informatik, Technische Universität Kaiserslautern, Gottlieb-Daimler-Straße, 67663 Kaiserslautern
Tel.: (06 31) 2 05-32 80
Fax: (06 31) 2 05-32 99
E-mail: buehmann@informatik.uni-kl.de

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 ist immer dann vorteilhaft, wenn Benutzer mehrere Male 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* [11], also dem Caching von statischen oder von der Anwendungslogik generierten Dokumenten oder deren Fragmenten, ist unabweisbar, 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 [6]. 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.

Anmerkung 1 Wir bewegen uns im Folgenden in der Welt der relationalen Datenbanksysteme mit SQL als Anfragesprache. Die generellen Prinzipien des Datenbank-Caching lassen sich jedoch auch auf andere Datenmodelle übertragen.

Im Gegensatz zu Ansätzen, denen die Wartung ziemlich starr definierter materialisierter Sichten zugrunde liegt [6], 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 einer Klasse von Datenbank-Caches ein, die auf Gleichheitsprädikaten basieren. In Abschnitt 2 auf der nächsten Seite 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*. Im darauf folgenden Abschnitt 3 stellen wir eine Verbesserung dieses Sondierungsverfahrens

vor, die eine flexiblere Nutzung des Cache-Inhalts erlaubt und das Phänomen des *negativen Caching* ans Licht bringt. Im Weiteren diskutieren wir die Eigenschaften dieses Verfahrens und mögliche Optimierungen. Wir zeigen, wie das Verfahren die Cache-Beschreibung zu vereinheitlichen hilft und welche neue Bedeutung es einer statischen Analyse der Cache-Struktur verleiht. Vor einer Zusammenfassung skizzieren wir schließlich in Abschnitt 4, wie man das Sondierungsverfahren einsetzen kann, um Daten aus dem Cache zu entfernen.

1.1 Cache Groups

Beim Constraint-basierten Datenbank-Caching bildet man einen Ausschnitt der Backend-Datenbank im Cache ab. Die statische Struktur dieses Ausschnitts wird in der Form von *Cache Groups* beschrieben.

Eine Cache Group besteht aus einer Menge von *Cache-Tabellen* sowie einer Menge von (Cache-)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 ihrer zugeordneten Backend-Tabelle.

Wir bezeichnen im Folgenden Cache-Tabellen mit Großbuchstaben (S, T, \dots) und ihre Spalten mit Kleinbuchstaben (e, q_1, \dots). Die diesen zugeordneten Backend-Tabellen und -Spalten tragen die gleichen Namen, ergänzt um einen Index B ($S_B, T_B, e_B, q_{1B}, \dots$). 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 [6; 8]. Unter der *Extension* eines Prädikats p verstehen wir dabei die Menge aller Sätze, die zur Auswertung von p benötigt werden.

1.2 Gleichheitsprädikate

Ein einfacher, aber nützlicher Typ von Cache Groups basiert auf Gleichheitsprädikaten [1; 3]. 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

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

für bestimmte Folgen $s_1, r_1, s_2, \dots, s_n$ von Spalten und ausgewählte Werte w haben. 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).

Anmerkung 2 Immer wenn wir von einem Prädikat als Anfrage sprechen, meinen wir eine PSJ-Anfrage, die sich nur auf die im Prädikat referenzierten Tabellen bezieht; die Projektion ist unwichtig und deswegen beliebig.

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

Um das Anfangsstück $T_1.s_1$ der Anfragen auswerten zu können, und als Mechanismus, um den Cache mit Daten zu füllen, führt man *Füllspalten* ein. Eine Füllspalte ist eine spezielle 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 Schema-Definition 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, nämlich 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 in einem Cache mehrere einfache Cache Groups parallel 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, führt man einen Typ von Constraints ein, 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 der *Quellspalte* $S.a$ zur *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 [7] – nähern wir uns kurz in Abschnitt 4. Hier konzentrieren wir uns auf die Beantwortung von Anfragen mit Hilfe von 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 Bearbeitung einer gegebenen Anfrage in zwei Schritten: Zunächst prüft man, ob die 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 den Cache auf die Existenz von Werten hin überprüfen, die in der Anfrage enthalten sind. Diesen Vorgang bezeichnen wir als *Sondierung* (englisch *probing*): Mit Hilfe einfacher Testanfragen wird die Beantwortbarkeit komplexerer Anfragen entschieden.

Bei Gleichheitsprädikaten des Typs $S.e = w$ genügt ein Test auf die Existenz von w in $S.e$, wenn aus der Existenz zusammen mit den Constraints abgeleitet werden kann, 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 (also auch 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 Quelltable des RCCs auf solche Sätze, die zu einer Prädikatsextension gehören, die sich im Cache befindet. 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 Quelltable 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 $S.a \rightarrow T.b$ 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.

Beispiel 1 Um zu illustrieren, wie eine Anfrage im Cache behandelt wird, betrachten wir die Konstellation in Abb. 1 auf der nächsten Seite; folgende Beispiele beziehen sich immer auf Ausschnitte daraus, wo nötig mit kleinen Änderungen. Die Legende in Abb. 2 auf der nächsten 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, \dots, O_B . Einige wenige von diesen (B, F, G, N, O) sind als Cache-Tabellen im Cache widerspiegelt (a). Die Spalten $F.f_1$ und $B.f_2$ sind als Füllspalten deklariert; fünf RCCs stellen die Beziehung zwischen den Cache-Tabellen her: $q_1 \rightarrow e, f_2 \rightarrow q_2, q_2 \rightarrow e, j \rightarrow g$ und $g \rightarrow h$. Die Belegung des Caches (c) zeigt den Zustand nach der Referenz der Füllspalten-Werte $f_1 \in \{A, B\}$ 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.)

Den Cache erreiche die PSJ-Anfrage eines Clients gegen die Tabellen F, N und M mit dem Prädikat $f_1 = A \wedge q_1 = e \wedge g = m$. (Wir nehmen an, dass 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 $f_1 = A$ 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 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.

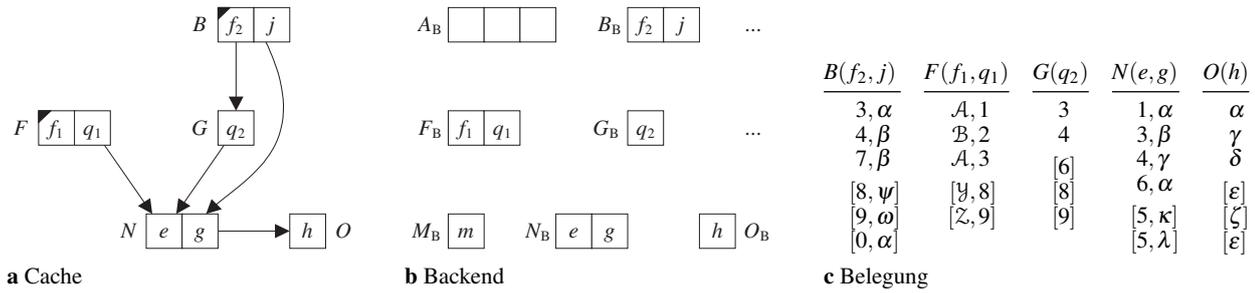


Abb. 1 (a) Cache-Tabellen repräsentieren eine Auswahl von (b) Backend-Tabellen im Cache und (c) enthalten eine Teilmenge derer Sätze (zusätzliche Sätze im Backend sind eingeklammert).



Abb. 2 Legende

Bislang wurde in Cache Groups zur einfachen Erkennung und Handhabung von Einstiegspunkten der Begriff der *Bereichsvollständigkeit* einer Spalte eingeführt [1; 5]. Eine Cache-Spalte heißt *bereichsvollständig*, wenn *jeder* Wert, der in dieser Spalte auftritt, wertvollständig ist. Deswegen genügt beim Sondieren ein einfacher 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. Einerseits wurde in bestimmten Spalten die Bereichsvollständigkeit erzwungen, indem diese Spalten explizit als so genannte *Cache Keys* [1] spezifiziert wurden; das sind Füllspalten mit maximaler Kandidatenmenge. Andererseits 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 [7; 8].

Anmerkung 3 Ein wichtiger Spezialfall bereichsvollständiger Spalten sind *Unique-Spalten*, also Spalten, die im Backend einen Unique-Constraint tragen. 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 (Minimalitätsannahme). Solche zusätzlichen RCCs wurden von uns *Optimierungs-RCCs* [8]

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 Abb. 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, strebte man bisher zwei Ziele an: Zum einen wollte man alle RCCs finden, die in der Cache Group gelten, um sie als Join-Richtungen verwenden zu können; zum anderen sollten alle bereichsvollständigen Spalten identifiziert werden. Diese potentiellen Einstiegspunkte spielten eine wichtigere Rolle, denn erst sie erlauben die Anwendung von RCCs.

Die Klassifizierung der Spalten der Cache Group in bereichsvollständige und nicht bereichsvollständige, die deswegen 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 bisher 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 all 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

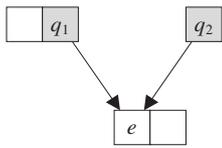


Abb. 3 Existenztest in den Quellspalten eingehender RCCs

vollständig sind. (In diesem Fall ist e nicht bereichsvollständig.)

Beispiel 3 Wir betrachten Abb. 3: Sie zeigt einen Ausschnitt aus Abb. 1a auf der vorigen Seite. 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 .

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.

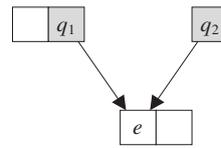
Beim Übergang vom alten zum neuen Sondierungsverfahren machen wir, um die Spalten für die Sondierung zu finden, einen Schritt zurück (vom potentiellen Einstiegspunkt entlang der eingehenden RCCs), aber keinen Rückschritt: Das Sondieren in den Quellspalten von RCCs hat mindestens zwei wichtige Vorteile 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, ist im Cache enthalten 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 kommt und in welchen nicht und wie sich die Sondierung in Quellspalten optimieren lässt, diskutieren wir in Abschnitt 3.3 auf der nächsten Seite.

3.1 Negatives Caching

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



Spalte	vorhandene Werte
q_1	1, 2, 3
q_2	3, 4
e	<u>1</u> , <u>3, 4</u> , 6
e_B	1, 3, 4, 5, 6

Abb. 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.)

- $e = 4$: 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.
- $e = 6$: 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.¹ 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.
- $e = 2$: In der Spalte e_B im Backend existiert der Wert 2 nicht, dementsprechend auch nicht in der Spalte e im Cache. 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 schon im Kontext des 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.” [2]

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 trifft das alte Sondierungsverfahren in solchen Fällen die Entscheidung, 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 über-

¹ Wie man Abb. 1 auf der vorhergehenden Seite entnehmen kann, war der RCC $j \rightarrow g$ dafür verantwortlich.

nehmen, nur um anschließend das leere Ergebnis zurückmelden zu können.

Beispiel 5 Wenn wir das alte Sondierungsverfahren auf die Situation aus Beispiel 4 in Abb. 4 anwenden, finden wir den Wert 2 nicht in Spalte e . Die Anfrage mit dem Prädikat $e = 2$ muss also im Backend ausgewertet werden; dort kommt der Wert 2 in Spalte e_B aber ebenfalls nicht vor.

Über einen erfolgreichen Einstieg in die Cache Group hinaus, bei der Ausnutzung von RCCs für Joins, ist das negative Caching schon im alten Modell des Sondierens und der Cache-Benutzung enthalten; bisher ist es aber unbemerkt geblieben.

Beispiel 6 Abb. 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 $j \rightarrow g$ aus Abb. 1a. Erst dann kann Spalte e als induziert bereichsvollständig erkannt werden und daher potentieller Einstiegspunkt sein.

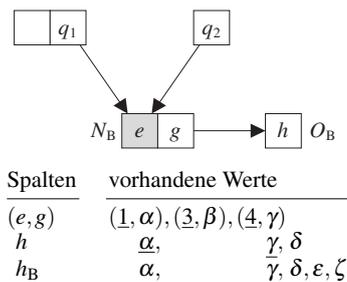


Abb. 5 Negatives Caching beim alten Sondierungsverfahren: Das Prädikat $e = 3 \wedge g = h$ führt im Cache zu einem leerem Ergebnis.

Die Sondierung in e zeigt, dass der Wert 3 im Cache vorhanden ist, also wegen der Bereichsvollständigkeit dort auch wertvollständig ist. Wegen des RCCs $g \rightarrow h$ 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 $(3, \beta)$, 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 oder drei

Schritte zurückgehen? Dies würde eine flexible Wahl der Sondierungsspalten ermöglichen.

Wir betrachten eine Kette von RCCs $a \rightarrow b \rightarrow c$ 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 $a \rightarrow c$? Ist also die durch eine Menge von RCCs definierte mathematische Relation über den Spalten transitiv?

Im Allgemeinen gilt dies nicht. – Wir stellen diese Frage bewusst nach der Betrachtung des negativen Caching: Dort wurde uns klar, 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 $a \rightarrow b$ wertvollständig in b , kann aber dort natürlich nicht existieren. Also macht der zweite RCC $b \rightarrow c$ 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 [1, 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. $c_1 \rightarrow c_2 \rightarrow \dots \rightarrow c_n \rightarrow c_1$.) 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 $c_i \rightarrow c_{i+1}$ die Wertvollständigkeit von v_j in der folgenden Spalte c_{i+1} bewirke. Dies und der ganze Satz lassen 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 ($n_e \geq 1$), 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.

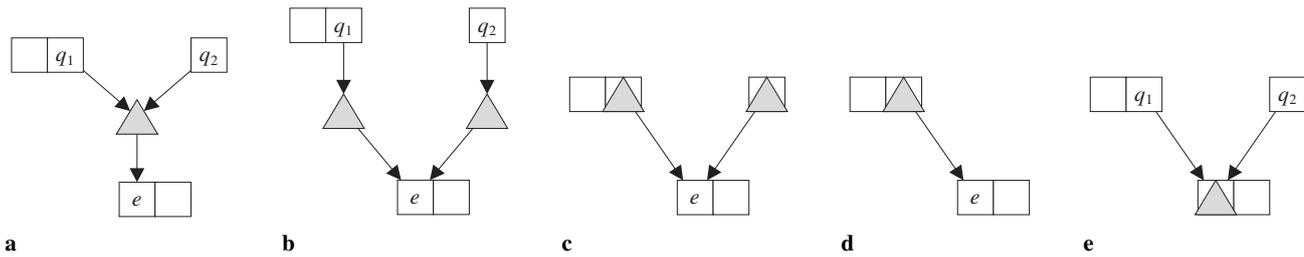


Abb. 6 Einsatz von Indexen: (a) Sondierung in Vollständigkeitsindex für e anstatt in den Quellspalten q_1 und q_2 ; (b) naives Sondieren mit normalen Indexen; (c) Abkürzung für (b); (d) nur eine Quellspalte: normaler Index ist Vollständigkeitsindex; (e) bei bereichsvollständigem e (siehe Beispiel 4)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

Anmerkung 4 Natürlich kann man während des Sondierens sofort abbrechen, wenn zum ersten Mal ein Existenztest erfolgreich abgeschlossen wird. Deswegen könnte man sich an dieser Stelle 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.

Anmerkung 5 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 Abb. 6a ist ein Vollständigkeitsindex (als Dreieck; siehe auch Abb. 2) für die Spalte e in der Cache Group dargestellt, die schon aus Abb. 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 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 Abb. 6c sind diese Indexe durch Überdeckung der durch sie indextierten Spalten dargestellt; dies kann als Abkürzung der in Abb. 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 (Abb. 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 Sondieren nach dem alten Verfahren direkt in der Spalte e (siehe Abschnitt 2) reiht sich nun in diesem Kon-

text als eine weitere mögliche Optimierung ein. Sie lässt sich nur dann anwenden, wenn man ableiten kann, dass e stets bereichsvollständig ist. Abb. 6e zeigt eine solche Situation in Anlehnung an Beispiel 4. 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 $f \rightarrow p$, 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 als Primärschlüssel 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.

Zusammenfassend gibt es 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 $k \rightarrow f_1$ von dieser

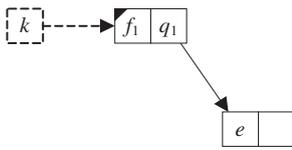


Abb. 7 Spalte f_1 ist Füllspalte. Eine Kontrolltabelle mit Spalte k und ein RCC $k \rightarrow f_1$ führen dazu, dass man zum Einstieg in f_1 bei der Sondierung genau so vorgehen kann wie bei jeder anderen Spalte.

Spalte auf die Füllspalte beheben. Allein die Einträge in dieser Kontrolltabelle bestimmen damit, welche Werte vollständig in die Füllspalte geladen werden (Abb. 7). Der zulässige Wertebereich von k ist genau die Kandidatenmenge von f_1 .

Bei genauerer Betrachtung behebt diese Umstrukturierung auch ein weiteres Problem: Bisher wurde keine Trennung vorgenommen von 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 solche Kontrolltabelle, die über ihren Inhalt die Belegung der eigentlichen Cache-Tabellen kontrolliert, taucht auch beim System MTCache auf [9; 10]. 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 der Effekt einer Art von RCC beschrieben, zumindest bei den Sichten im angegebenen Beispiel [9, Abschnitt 5].

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.

- Erneut 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 [5], wird also ein Algorithmus zur Bestimmung aller Optimierungs-RCCs einer Cache Group benötigt (vielleicht auf der Grundlage eines Kalküls).

Anmerkung 6 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 Abb. 8a zeigt eine Cache Group in einer Form, wie man sie spezifiziert haben könnte, mit einer Füllspalte und fünf explizit angegebenen RCCs. In Abb. 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.

4 Entfernung von Prädikatsextensionen

Wir betrachten eine zusätzliche Anwendung des neuen Sondierungsverfahrens, nämlich das Entfernen von Prädikatsextensionen aus dem Cache: Beim normalen Betrieb eines

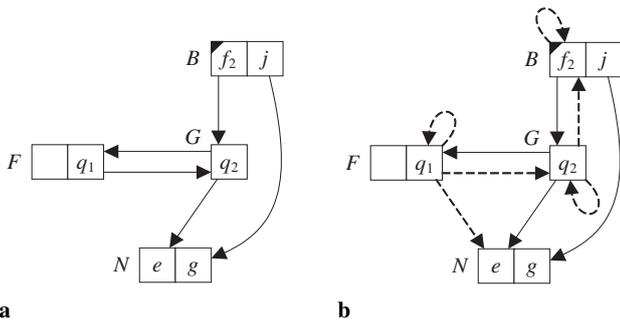


Abb. 8 Optimierungs-RCCs: (a) Eine Cache Group mit spezifizierten RCCs, (b) dieselbe mit abgeleiteten und als redundant erkannten RCCs (gestrichelt).

Datenbank-Caches werden ausgehend von den Kontrolltabelle Prädikatsextensionen in den Cache gefüllt, die vermutlich in nächster Zukunft gebraucht werden. Um den Umfang der im Cache gehaltenen Extensionen in Grenzen zu halten und damit auch den Aufwand, diese Extensionen bei Veränderungen im Backend zu warten, ist es notwendig, Extensionen entfernen zu können.

Die Entfernung von Extensionen ist nicht trivial: Mehrere Extensionen, die von verschiedenen Kandidatenwerten in den Kontrolltabellen abhängen, können sich je nach Struktur der Cache Group mehr oder weniger stark überlappen. Sie benutzen dann bestimmte Satzmenngen gemeinsam.

Definition 3 Ein Satz t in einer Cache-Tabelle T heißt *direkt abhängig* von einem Satz s in einer Cache-Tabelle S , wenn ein RCC $S.a \rightarrow T.b$ existiert und s und t in den beteiligten Spalten a und b denselben Wert tragen.

Die transitive Fortsetzung dieser Relation verhilft uns in einfacher Weise zum Begriff der (allgemeinen) *Abhängigkeit* von Sätzen untereinander. Jeder Satz (und damit jeder Kandidatenwert) in einer Kontrolltabelle identifiziert so eine bestimmte Prädikatsextension, nämlich die Menge der von ihm abhängigen Sätze. Entsprechend kann ein Satz von mehreren Sätzen in einer oder in mehreren Kontrolltabellen abhängig sein.

Der radikalste und offensichtlich kaum erwünschte Weg, solche Extensionen zu entfernen, besteht darin, den gesamten Inhalt des Caches zu löschen. Schließlich sind die Daten des Caches redundant – die folgenden Anfragen können zunächst aus dem Backend beantwortet werden –, und ein leerer Cache erfüllt alle definierten Constraints.

Um in kleineren Schritten und damit flexibler Speicherplatz freigeben zu können, sollte es möglich sein, Extensionen einzeln oder in kleineren Gruppen zu entfernen. Dabei ist darauf zu achten, keine andere Extension und damit Cache-Constraints zu verletzen. Es dürfen also nur solche Sätze einer Extension entfernt werden, die nicht auch zu einer anderen Extension gehören, die im Cache verbleiben soll. Um diese Sätze zu identifizieren, können wir das neue Sondierungsverfahren heranziehen.

Betrachten wir eine bestimmte Spalte $S.e$ im Cache, die als Einstiegspunkt für ein Gleichheitsprädikat $S.e = w$ dienen soll. Ein erfolgreiches Sondieren im Cache sagt uns, dass sich die Extension dieses Prädikats im Cache (in Tabelle S) befinden muss, um die Cache-Constraints sicherzustellen. Umgekehrt muss sich ein Satz *nicht* im Cache befinden, wenn die Sondierungen für seine Spaltenwerte fehlschlagen; er wird dann von keiner Extension benutzt. Denn außer den RCCs erzwingt kein Mechanismus die Anwesenheit von Sätzen in Cache-Tabellen. Ein Satz aber, der sich nicht im Cache befinden muss, kann gelöscht werden.

Anmerkung 7 Wohlgermerkt, für Sondierungen zu berücksichtigen sind alle Spalten der betroffenen Cache-Tabelle, hier S , die Einstiegspunkte sein können, nicht nur $S.e$; das sind beim neuen Sondierungsverfahren alle Spalten mit einem eingehenden RCC.

Mit dieser zentralen Überlegung ist folgendes Vorgehen zum Entfernen der Extension möglich, die von einem Wert $K = w$ einer Kontrolltabelle K abhängt:

1. Entferne den Wert w aus der Kontrolltabelle K .
2. Solange Sätze im Cache existieren, für die jede mögliche Sondierung fehlschlägt, entferne sie aus dem Cache.

Dabei müssen in Schritt 2 nicht alle Sätze im Cache betrachtet werden: Es genügt, diejenigen zu überprüfen, die direkt von einem schon entfernten Satz abhängen. Denn nur in diesen Fällen kann die Sondierung überhaupt fehlschlagen: Vor dem Beginn der Entfernung ist der Cache in einem gültigen Zustand; für jeden Wert, der sich im Cache befindet, muss mindestens eine Sondierung dies bestätigen; sonst wäre er nicht im Cache (unter der Minimalitätsannahme; vgl. Abschnitt 2.2).

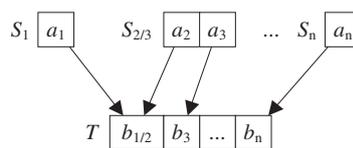


Abb. 9 Entfernung von Extensionen: Die Sondierung kann entscheiden, welche Sätze aufgrund der Cache-Constraints in einer Cache-Tabelle T verbleiben müssen.

In zyklensfreien Cache Groups kann man so ausgehend von der Kontrolltabelle in topologischer Sortierordnung bezüglich der RCCs über die Cache-Tabellen iterieren. Die für jede Cache-Tabelle T durchzuführenden Sondierungs- und Löschoperationen lassen sich am elegantesten durch eine SQL-Anweisung beschreiben. Wir nehmen an, T habe die eingehenden RCCs $S_i.a_i \rightarrow T.b_i$ für $i = 1, \dots, n$, wobei verschiedene Indizes nicht zwingend verschiedene Spalten oder Tabellen bezeichnen (Abb. 9). Dann löscht folgende Anweisung alle Sätze, die nicht mehr im Cache sein müssen:

```
delete from T
where not exists (select * from S1 where a1 = b1)
and not exists (...)
and not exists (select * from Sn where an = bn)
```

Ein Problem tritt auf, wenn es zyklische Abhängigkeiten zwischen Sätzen gibt: In einem solchen Fall ist die Sondierung entlang des gesamten Zyklus erfolgreich, selbst wenn sämtliche beteiligten Sätze entfernt werden könnten, ohne die Cache-Constraints zu verletzen. Das liegt daran, dass die Sondierung sich nur auf lokale Informationen stützt (einen Schritt zurück) und nicht die weitere Herkunft von Sätzen berücksichtigt (etwa den Pfad zurück bis zu einer Kontrolltabelle).

Wechselseitige Abhängigkeiten zwischen Sätzen können prinzipiell nur dann auftreten, wenn in der Cache Group selbst ein RCC-Zyklus existiert; für solche Fälle, in denen oft auch das Laden unkontrolliert erfolgen kann [8], müsste man dann zu anderen Entfernungsstrategien greifen. Hier ist auch das Entfernen von Extensionen kaum beherrschbar – zumindest mit Hilfe der einfachen Sondierung nicht realisierbar.

Interessant ist die Frage, ob die von einem Kandidatenwert einer Kontrolltabelle abhängigen Sätze allesamt in einem Schritt (in einer Transaktion) gelöscht werden müssen oder ob Teilmengen für eine gewisse Zeit im Cache verbleiben können, ohne Cache-Constraints zu verletzen. Solche Teilmengen können offensichtlich dann im Cache bleiben, wenn die grundlegendere Wertvollständigkeit am Ende von RCCs nicht zerstört wird. (Das wäre beispielsweise gesichert, wenn man, wie oben angedeutet, alle zu löschenden Sätze einer Cache-Tabelle gemeinsam entfernte.) Allerdings ist damit die Minimalitätsannahme nicht mehr erfüllt. Somit können diejenigen Optimierungs-RCCs, bei deren Ableitung diese Annahme eine Rolle gespielt hat, nicht weiterverwendet werden.

5 Zusammenfassung und Ausblick

Die effektive Nutzung des Inhalts eines Constraint-basierten Datenbank-Caches beginnt bei der Wahl eines auf die benutzten 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 potentiellen 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 Anfragetypen aus dem Cache beantwortet werden als vorher. Bei Anfragen, die sich auf leere Satzmenge beziehen, 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 die direkten 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, etwa entlang einer Primär-/Fremdschlüssel-Beziehung. 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. So ergibt sich auch bei der Durchführung der Sondierung eine große Flexibilität.

Die Kenntnis aller Optimierungs-RCCs einer gegebenen Cache Group ist nach Einführung des neuen Sondierungsverfahrens 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 Constraints, die diese unterstützen, stellen einen sehr einfachen Fall des Constraint-basierten Caching dar. Eine offene Frage ist, durch welche Constraints andere Prädikatstypen unterstützt werden können, und wie diese mit den bestehenden wechselwirken, falls sie simultan in einem Cache erfüllt werden sollen. Schon die Überlagerung mehrerer einfacher (und für sich harmloser)

Cache Groups kann durch RCC-Zyklen ein 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, verschiedene Strategien zur Bearbeitung und Propagierung von Änderungen im Cache und im Backend sowie die effiziente Entfernung von gegebenenfalls überlappenden Prädikatsextensionen aus dem Cache. Für die Entfernung haben wir in einfachen Situationen den Bezug zum Sondierungsverfahren gezeigt. Weiter reichende Fragestellungen richten sich beispielsweise auf die Auflösung der starren Rollenverteilung Cache/Backend und auf die Möglichkeit, nicht nur den Inhalt, sondern auch die Struktur einer Cache Group selbst an ein verändertes Anfrageprofil anzupassen.

Danksagung Philipp Dopichaj hat frühere Versionen dieses Artikels gelesen und mir viele wertvolle Hinweise gegeben; Gleiches gilt für die unbekanntenen Gutachter. Ich danke ihnen allen sehr.

11. Vossen G, Rahm E (eds) (2002) Web & Datenbanken: Konzepte, Architekturen, Anwendungen, dpunkt.verlag, Heidelberg, chap 7, pp 191–216

Literatur

1. Altinel M, Bornhövd C, Krishnamurthy S, Mohan C, Pirahesh H, Reinwald B (2003) Cache tables: Paving the way for an adaptive database cache. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, Morgan Kaufmann, pp 718–729
2. Andrews M (1998) Negative caching of DNS queries (DNS NCACHE). Request for Comments (RFC) 2308, URL <ftp://ftp.rfc-editor.org/in-notes/rfc2308.txt>
3. Bornhövd C, Altinel M, Mohan C, Pirahesh H, Reinwald B (2004) Adaptive database caching with DBCache. Data Engineering Bulletin 27(2):11–18
4. Bühmann A (2005) Einen Schritt zurück zum negativen Datenbank-Caching. In: BTW 2005: Datenbanksysteme für Business, Technologie und Web, Gesellschaft für Informatik, Karlsruhe, Lecture Notes in Informatics
5. Härder T, Bühmann A (2004) Database caching – Towards a cost model for populating cache groups. In: Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Springer, Lecture Notes in Computer Science, vol 3255, pp 215–229
6. Härder T, Bühmann A (2004) Datenbank-Caching – Eine systematische Analyse möglicher Verfahren. Informatik – Forschung und Entwicklung 19(1):2–16
7. Härder T, Bühmann A (2004) Query processing in constraint-based database caches. Data Engineering Bulletin 27(2):3–10
8. Härder T, Bühmann A (2004) Value complete, domain complete, predicate complete – Magic words driving the design of cache groups, URL <http://www.dvs.informatik.uni-kl.de/pubs/papers/HB04.Magic.html>
9. Larson P, Goldstein J, Guo H, Zhou J (2004) MTCache: Mid-tier database caching for SQL server. Data Engineering Bulletin 27(2):35–40
10. Larson P, Goldstein J, Zhou J (2004) MTCache: Transparent mid-tier database caching in SQL server. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, IEEE Computer Society, pp 177–189