Diplomarbeit

Semantisches Caching in ontologiebasierten Mediatoren

Marcel Karnstedt
marcel@karnstedt.com

Betreuer:
PD Dr.-Ing. habil. K.-U. Sattler

Martin-Luther-Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Institut für Informatik
D–06120 HALLE (Saale)

1. Oktober 2003

Datenintegration stellt durch die vielen autonomen und heterogenen Quellen im World Wide Web (WWW) immer noch eine große Herausforderung dar. Ein Ansatz, die entstehenden Probleme zu lösen, ist die Nutzung eines Mediators, der das explizite Domänenwissen in Form eines Vokabulars oder einer Ontologie zur Integration und Anfragebearbeitung modelliert. An die Entwicklung solcher ontologiebasierter Mediatorsysteme ergeben sich hohe Anforderungen, um eine effiziente Ausführung der Anfragebearbeitung zu gewährleisten. Ein globaler Cache kann dabei die Antwortzeit und die Auslastung der Quellsysteme deutlich reduzieren. In dieser Arbeit wird ein semantischer Cache für das YACOB-Mediatorsystem entwickelt und implementiert. Der Cache ist eng an das Konzeptmodell des Mediators geknüpft und gruppiert die Daten in semantische Regionen. Die Daten werden nur anhand ihrer semantischen Beschreibung identifiziert. Wesentliche Bestandteile der Arbeit bilden die detaillierte Beschreibung der Anfragebearbeitung, die Schilderung der Cache-Struktur und die Evaluation des entwickelten semantischen Caches.

Inhaltsverzeichnis

1 Einleitung

Anwendungen, die einen integrierten Zugriff auf heterogene Datenbestände bieten, gewinnen mit wachsender Größe und Nutzung des World Wide Web immer mehr an Bedeutung. Gerade dort entstehen dabei größte Herausforderungen, denn es bietet durch seinen unabhängigen und dynamischen Charakter ein hohes Maß an Autonomie und Heterogenität in den unterschiedlichen Quellen. Weiterhin treten Probleme wie Skalierbarkeit und Evolutionsfähigkeit hinsichtlich einer großen Anzahl teilweise noch wechselnder Quellen in besonderem Maße zu Tage. Lösungsansätze zur Datenintegration reichen hierbei von einfachen (Meta-)Suchmaschinen über materialisierende Ansätze bis hin zu Mediatorsystemen. Letztgenannte Systeme zerlegen Anfragen auf einem globalen Schema in Teilanfragen, leiten diese an die verschiedenen Quellsysteme zur Beantwortung weiter und führen die Teilergebnisse zu einer Gesamtheit zusammen. Neuere Ansätze modellieren dazu explizites Hintergrundwissen in Form semantischer Metadaten als Vokabular, Begriffs- bzw. Konzepthierarchie oder gar Ontologie[1]. Im letzten Fall spricht man von ontologiebasierten Mediatoren. Ein Verteter dieser Klasse ist der YACOB-Mediator, welcher Domänenwissen in Form von Konzepten und deren Beziehungen modelliert. Dieser Mediator wurde für die integrierte Suche in kulturhistorischen Datenbanken entwickelt, die Informationen über kriegsbedingt verlorengegangene Kulturgüter verwalten, und bildet die Grundlage dieser Arbeit.

Allen Konzepten zur virtuellen Datenintegration gemein ist das Ziel einer global effizienten Ausführung, im Falle des World Wide Web insbesondere beeinträchtigt durch die Integration von Datennetzen mit begrenztem Durchsatz. Effizienz umfasst in diesem Fall einerseits eine angemessene Antwortzeit für Nutzer des Systems sowie andererseits die Auslastung der integrierten Quellen durch redundante oder irrelevante Anfragen zu minimieren. Gerade im YACOB-Mediator ist dabei die Möglichkeit einer interaktiven Anfrageformulierung mit schrittweiser Verfeinerung von besonderer Bedeutung. Vor allem das Erreichen akzeptabler Antwortzeiten ist besonders problematisch, nicht umsonst ist WWW weit verbreitet als Synonym für „World Wide Wait“ bekannt.

Eine Möglichkeit, einen effizienten Ablauf zu gewährleisten und die Skalierbarkeit des Systems zu erhöhen, bietet die Zwischenspeicherung von ausgeführten Anfragen inklusive der entsprechenden Ergebnisse in einem *Cache*. In einem Cache werden nach einer festgelegten Nachladestrategie Kopien der eigentlichen Daten erfasst. Diese können aus dem Cache schneller gelesen werden als von den Datenquellen. Ein Cache dient somit als schneller Zwischenspeicher zur Beschleunigung des Datenzugriffs, seine physische Kapazität ist im Vergleich zu denen der Datenquellen in der Regel gering. Ein bekanntes Cache-Verfahren ist z.B. das Seiten-Caching von Hauptspeicherdaten, bei dem Kopien von angefragten Seiten des Hauptspeichers im Cache zwischengespeichert werden. Caching-Techniken spielen eine wichtige Rolle in Betriebssystemen, Datenbanksystemen und verteilten Dateisystemen.

[1] Man unterscheidet den philosophischen (Die Lehre vom Sein) und den durch die Informatik geprägten Begriff. In diesem Fall steht er für das Darstellungsmodell eines Weltausschnittes, das Domänenwissen durch ein einheitliches Vokabular auf Basis von Konzepten modelliert - hier in Form eines semantischen Netzwerkes: ein Graph, dessen Knoten Konzepte oder individuelle Objekte darstellen und die Kanten die Beziehungen oder Assoziationen zwischen diesen Objekten repräsentieren.

Im Zusammenhang mit dem Web bedeutet Caching, dass wichtige, d.h. häufig oder in jüngster Vergangenheit benutzte Daten, als temporäre Kopien auf Rechnern lokal gespeichert werden, die „näher“ an den anfordernden Clients liegen. Dabei ist „Nähe“ nicht unbedingt örtlich zu verstehen, sondern kann sich z.B. auf Transferraten beziehen, was auch geographisch weit entfernte Rechner für die Zwischenspeicherung attraktiv machen kann. Will ein Client auf ein Datenobjekt zugreifen, so wird zuerst geprüft, ob dieses Objekt in einem Cache verfügbar ist, bevor es von seinem Originalspeicherplatz angefordert wird. Ein solcher *Cache-Treffer* reduziert die Zugriffszeit auf das Datenobjekt signifikant. Liegt kein entsprechendes Objekt im Cache vor, so spricht man von einem *Cache-Fehler*. Unter Umständen kann es auch sinnvoll sein, Daten schon in den Cache zu laden, bevor sie das erste Mal angefragt werden. Dieses Verfahren nennt sich *Vorladen* (*Prefetching*) und wird vom Caching im engeren Sinn abgegrenzt. Weiterhin ist das Caching von der Technik des *Spiegelns* (*Mirroring*) zu unterscheiden, bei dem es sich um eine meist längerfristige Datenplatzierungsmaßnahme auf sogenannten „Mirror-Sites“ handelt. Caching, Vorladen und Spiegeln werden aber allesamt als Techniken der *dynamischen Datenreplikation* eingestuft. Typischerweise replizierte Datenobjekte sind im Web z.B. statische oder dynamische HTML-Seiten, Bilder und Videos, aber auch Datenbankanfrageresultate.

Als geeignetes Mittel zur Erfüllung der Anforderungen an einen Cache in ontologiebasierten Mediatorsystemen hat sich das semantische Caching erwiesen. Dabei werden Daten zusammen mit beschreibenden (Meta-)Daten gespeichert, im vorliegenden Fall die von den Quellen erhaltenen Ergebnisdaten und die zugehörigen Anfragen, die diese Ergebnisdaten beschreiben. Folgende Anfragen, bei schrittweiser Anfrageverfeinerung Ergebnismengen einschränkende oder Ergebnismengen erweiternde, können dann ganz oder teilweise mit Hilfe der zwischengespeicherten Ergebnisse beantwortet werden.

In dieser Arbeit wird das Konzept eines semantischen Caches für den erwähnten YACOB-Mediator entwickelt und implementiert. Der Cache ist eng an die im Mediatorsystem definierte Ontologie gekoppelt. Wesentliche Punkte bilden die Diskussion verschiedener Aspekte der Struktur und Verwaltung des Caches und die detaillierte Beschreibung der Anfragebearbeitung. Abschließende Experimente und Tests dienen zur Bewertung des realisierten Caches.

In Kapitel 2 wird zunächst ein vertiefender Einblick in die Thematik ontologiebasierter Mediatoren im Allgemeinen gegeben und der YACOB-Mediator als spezieller Vertreter näher vorgestellt. Daraufhin wird Bezug auf Notwendigkeit und Zweck semantischen Cachings in Mediatorsystemen sowie die prinzipielle Umsetzung genommen. Im darauf folgenden Kapitel werden die Grundlagen der Problematik und existierende Lösungsansätze erläutert. Ein Lösungsvorschlag für einen Grundbestandteil des Cachings, das *Query Containment* beschränkt auf Prädikate, bildet einen eigenen Abschnitt. In Kapitel 4 wird auf Aspekte der Struktur und Verwaltung eingegangen, darauf folgt die Beschreibung der Anfragebearbeitung im Cache und ihrer Integration in den globalen Ablauf. Nach kurzen Erläuterungen zur Implementierung bildet ein Abschnitt mit Experimenten und Simulationen zur Evaluation des Caches in Kapitel 6 den letzten behandelten Punkt, bevor die Arbeit mit einer Zusammenfassung des Geleisteten und einigen ausblickenden Gedanken in Kapitel 7 abgeschlossen wird.

2 Ontologiebasierte Mediatoren

2.1 Mediatorsysteme im Überblick

Im Angesicht der riesigen Datenmenge im WWW ist es oftmals schwierig, relevante Inhalte zu identifizieren. So liefern z.B. Suchmaschinen auf eine Anfrage meist tausende Dokumente, die den Suchbegriff an irgendeiner Stelle enthalten. Lösung bietet die Unterstützung strukturierter Anfragen, d.h. Anfragen über konkrete, identifizierbare Eigenschaften von Objekten, wie etwa Preis oder Bezeichnung. Zur Beantwortung solcher Anfragen sind verschiedene, für den gegebenen Anwendungsbereich relevante Quellen zu integrieren, d.h. einerseits eine einheitliche Sicht bezüglich Schnittstellen und Struktur all dieser Quellen bereitzustellen und andererseits für Transparenz hinsichtlich der Herkunft der Daten zu sorgen. Dabei besteht eine besondere Anforderung darin, die Heterogenität der Quellen zu überwinden. Diese äußert sich auf verschiedenen Ebenen: z.B. auf Systemebene durch unterschiedliche Anfrageschnittstellen oder auf Datenmodellebene durch die Verwendung von HTML, XML, relationaler oder objektorientierter Schemata. Besonderheit für die Datenintegration im Web ist die meist große Anzahl häufig wechselnder autonomer Quellen, bei denen es sich oft um keine vollfunktionalen Datenbanksysteme handelt und über deren Eigenschaften in der Regel wenig Informationen vorliegen. In [SCS02] werden Systeme zur Datenintegration im Web klassifiziert und die Unterschiede zu heterogenen Datenbanksystemen vertieft betrachtet. Im Besonderen werden Mediatoren und die aus ihrer Nutzung zur Datenintegration resultierenden Aufgaben und Probleme detailliert beschrieben und verschiedene bekannte Systeme vorgestellt.

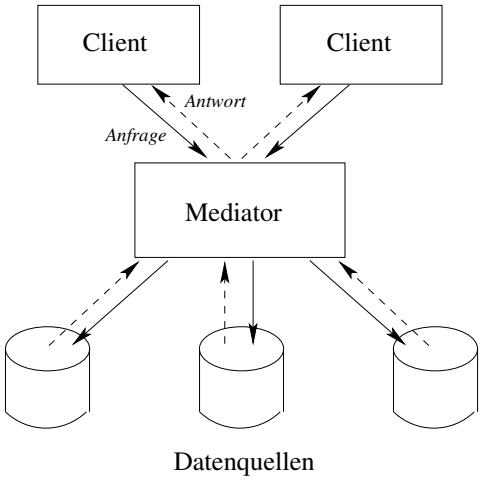


Abbildung 1: Mediator-Architektur nach Wiederhold

Mediatorsysteme werden dem virtuellen Ansatz zur Datenintegration zugeordnet, bei dem Sichten über die Quelldaten definiert werden. Die Idee von Mediatoren geht auf Wiederhold ([Wie97]) zurück und bezeichnet in einer sehr allgemeinen Form eine Softwarekomponente zur Vereinfachung, Reduzierung, Kombination und Erklärung von Daten.

In Abbildung 1 ist die Positionierung des Mediators in einer Architektur aus drei Ebenen abgebildet. Dabei werden in der Mediatorebene, im Gegensatz z.B. zum Data-Warehouse-Ansatz, keine Nutzdaten gespeichert. Anfragen an den Mediator sind somit immer durch Anfragen an die Quellsysteme zu beantworten. Das erhöht die Aktualität der Daten und verhindert eine redundante Datenhaltung. Demgegenüber stehen die Nachteile einer aufwändigen Anfrageausführung und die Abhängigkeit von der Verfügbarkeit der Quellen. Mediatorsysteme ermöglichen also einen effizienten und aktuellen Zugriff auf weltweit verteilte Daten sowie deren Kombination. Ihre Aufgabe besteht darin, strukturierte Anfragen zu beantworten, indem Daten aus externen, heterogenen und eventuell häufig wechselnden Quellen extrahiert und kombiniert werden. So könnten z.B. auf eine Anfrage zu einem bestimmten Autor alle seine Werke aus einer Literaturdatenbank ermittelt, persönliche Informationen aus einer Online-Enzyklopädie bestimmt und eventuell noch verfügbare Rezensionen aus einer dritten Datenquelle gesammelt werden.

In Mediatorsystemen werden dem Nutzer Daten der vielen unterschiedlichen und oft wechselnden Quellen über ein globales Schema zugänglich gemacht. Anfragen, die auf diesem globalen Schema formuliert sind, werden in Teilanfragen zerlegt und transformiert, die den lokalen Schemata der Quellen entsprechen. Diese werden von den einzelnen Quellen beantwortet und die erhaltenen Teilergebnisse zu einer Gesamtheit zusammengefasst. Das dem Anwender präsentierte Gesamtergebnis entspricht wiederum dem globalen Schema auf dem die Anfrage formuliert wurde. Dabei werden nur Leseoperationen (Anfragen) unterstützt und meist ein semistrukturiertes Datenmodell verwendet.

Man kann Mediatoren grob nach der Gestaltung der Korrespondenzen zwischen den lokalen Schemata der Quellen und dem globalen Mediatorschema unterscheiden. Vorliegende Ansätze sind der Global-as-View- (GaV) und der Local-as-View-Ansatz (LaV). Beim GaV-Ansatz wird das globale Schema als Sicht über den lokalen Schemata definiert, umgekehrt wird beim LaV-Ansatz von einem globalen Schema ausgegangen, über das die lokalen Schemata als Sichten definiert werden. Der GaV-Ansatz hat den Vorteil einer einfacheren Anfrageverarbeitung, beim LaV-Prinzip ist das Hinzufügen bzw. Entfernen von Quellen deutlich einfacher, da hier keine Korrespondenzen zwischen Quellen berücksichtigt werden müssen.

Bei Mediatorsystemen der ersten Generation erfolgt die Integration im Wesentlichen auf struktureller Ebene. Dabei werden die Daten der unterschiedlichen Quellen aufgrund von strukturellen Korrespondenzen, wie z.B. gemeinsamen Attributen oder der Zugehörigkeit zu gleich strukturierten Klassen, kombiniert. Neuere Ansätze verwenden stattdessen semantische Metadaten in Form von Vokabularen, Begriffs- bzw. Konzepthierarchien oder gar Ontologien um das Hintergrundwissen explizit zu modellieren. Diese Metadaten werden für Anfrageverarbeitung und Datenintegration genutzt. Ähnliche Ansätze semantisches Wissen zu nutzen finden sich im Rahmen des *Semantic Web*, hier um die (wissensbasierte) Verarbeitung von Web-Dokumenten zu erleichtern. Erste Ergebnisse sind Modelle und Sprachen für Vokabulare und Ontologien, wie z.B. RDF Schema (RDFS), auf welchem auch das Konzeptmodell des im Folgenden betrachteten YACOB-Mediatorsystems beruht. Als besondere Anforderung aus Sicht der Datenintegration ergibt sich die Notwendigkeit, den Bezug zu Quelldaten herzustellen.

Informationen darüber, wie die Quellsysteme ein gegebenes Konzept der semantischen Ebene strukturell und inhaltlich realisieren, müssen müssen für jede Quelle vorliegen und in die Anfragetransformation integriert werden.

Im folgenden Abschnitt wird ein auf dem LaV-Prinzip basierender Mediator, der YACOB-Mediator vorgestellt.

2.2 Das YACOB-Mediatorsystem

Im YACOB-Mediatorsystem wird das explizite Domänenwissen durch Konzepte, ihren Eigenschaften und Beziehungen in Form einer Ontologie modelliert. Die in diesem Metamodell definierten Konzepte fungieren als begriffliche „Anker“ für die Datenintegration. Zu den definierten Konzepten existieren Extensionen (eine Menge von Instanzen) in den einzelnen Quellsystemen. Entwickelt wurde das System für den integrierten Zugriff auf Web-Datenbanken, die Informationen über im zweiten Weltkrieg verlorengegangene oder gestohlene Kultugüter verwalten. Beispiele solcher Quellen, die auch schon in das YACOB-System integriert wurden, sind `www.lostart.de`, `www.herkomstgezocht.nl` und `www.restitution-art.cz`.

Das Zwei-Ebenen-Modell

Die im Metamodell definierten Konzepte bilden die Meta- oder Konzeptebene, welche die Semantik der Daten und deren Beziehungen beschreibt. Diese Konzepte werden mit Informationen zur Abbildung auf die Quelldaten verbunden. Die in den Quellsystemen gespeicherten Daten werden auf auf einer zweiten Ebene, der Daten- oder Instanzebene, in XML repräsentiert.

Ein einfaches semistrukturiertes Modell auf Basis von XML bildet das Datenmodell für die Instanzebene. Alle Daten im Mediator und im Austausch mit den Quellsystemen bzw. deren *Wrappern* werden somit in XML repräsentiert. Wrapper bilden dabei die Verbindung zwischen Mediator und Quellsystemen. Sie bieten dem Mediator einen einheitlichen Zugriff auf die Quellen, zu ihren Aufgaben gehört demnach die Abbildung zwischen dem Datenmodell des Mediators und dem spezifischen Modell der Quelle sowie die Übersetzung der vom Mediator initiierten Anfrage. Für die Formulierung von Anfragen auf der Instanzebene wird eine eingeschränkte Untermenge von XPath verwendet. Jede zu integrierende Quelle muss also in der Lage sein, Anfragen in XPath zu verarbeiten und zugehörige Daten in XML zu exportieren. Notwendige Transformationen werden automatisch durchgeführt. So kann jede Quelle XML-Daten mit einer beliebigen DTD liefern.

Die Definition des eingesetzten Modells zur Beschreibung der Konzeptebene basiert auf RDF Schema (RDFS). RDF (*Resource Description Framework*) ist ein vom W3C entwickelter Mechanismus zur Erzeugung und zum Austausch von Metadaten für Web-Dokumente. RDFS definiert auf diesem graphbasierten Modell weitere Primitive, wie *Class*, *Property* oder *subClassOf*, und ermöglicht damit die Beschreibung einfacher Vokabulare oder Ontologien.

In Abbildung 2 ist ein Teil des verwendeten Konzeptschemas abgebildet.

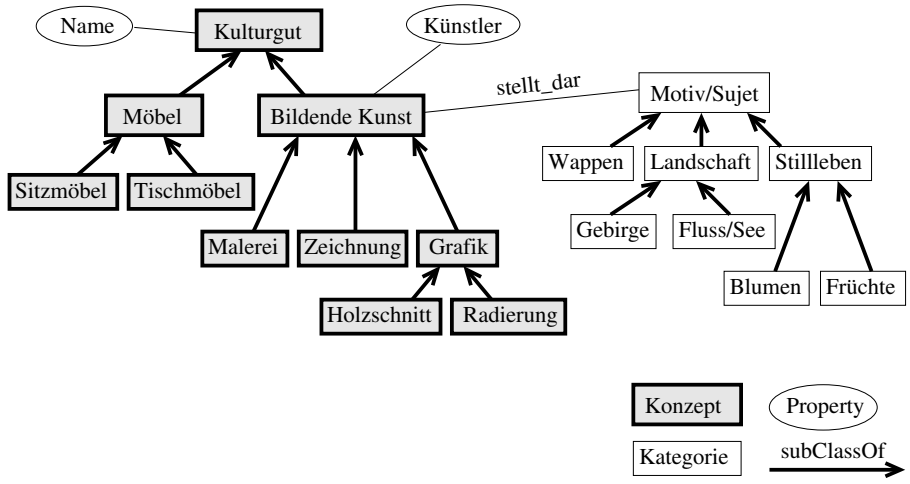


Abbildung 2: Konzepthierarchie

Ein Konzept im Mediatormodell entspricht einer Klasse in RDFS, genauer einer Subklasse einer speziellen Klasse *Concept*. Eigenschaften von Konzepten und Beziehungen untereinander werden als *Properties* abgebildet. Dabei werden ebenfalls *Properties* verwendet um Beziehungen zwischen Konzepten zu modellieren, die über die bereits in RDFS definierten Beziehungen wie *subClassOf* hinausgehen. Als Werte- und Bildbereich solcher *Properties* sind dann aber nur Klassen zugelassen. Ein Beispiel dafür sind die abgebildeten Kategorien, spezielle Varianten einer RDFS-Klasse. Eine Kategorie kann als Begriff angesehen werden, der in verschiedenen Quellen durch unterschiedliche Werte repräsentiert wird. Kategorien weisen im Gegensatz zu Konzepten keine Extension auf und werden eingesetzt, um semantisch zusammenhängende Objekte zu gruppieren. In der Hierarchie können Kategorien unter Verwendung der *subClassOf*-Beziehung organisiert werden.

Ein Quellsystem unterstützt ein bestimmtes Konzept, wenn es eine Teilmenge der Extension bereitstellt. Dabei müssen nicht alle definierten Eigenschaften des Konzeptes unterstützt werden. Um auf dem Mediatorschema formulierte Anfragen auf die Schemata der Quellen abbilden zu können müssen Abbildungsinformationen im Mediator erfasst werden. Konkret sind Angaben nötig, wie die einzelnen Quellen Daten zu einem gegebenen Konzept liefern und wie deren Struktur auf die durch das Konzept und dessen Eigenschaften definierte globale Struktur abgebildet wird. Diese Informationen werden ebenfalls genutzt um die erhaltenen Ergebnisdaten wieder in das globale Schema zu überführen. Die Quellenschemata werden dabei als Sichten auf dem globalen Mediatorschema definiert, im YACOB-Mediator wird also das LaV-Prinzip umgesetzt.

Abbildung 3 zeigt einen Ausschnitt des eingesetzten Konzeptschemas inklusive der Abbildungsinformationen. Für die einzelnen Elemente des Konzeptschemas sind verschiedene Abbildungsvorschriften festgehalten. Dabei wird auch die Spezialisierungshierarchie genutzt. So muss nicht zu jedem Konzept eine Vorschrift definiert werden, sondern nur für die Konzepte, die bezüglich einer Quelle Blätter der Hierarchie darstellen.

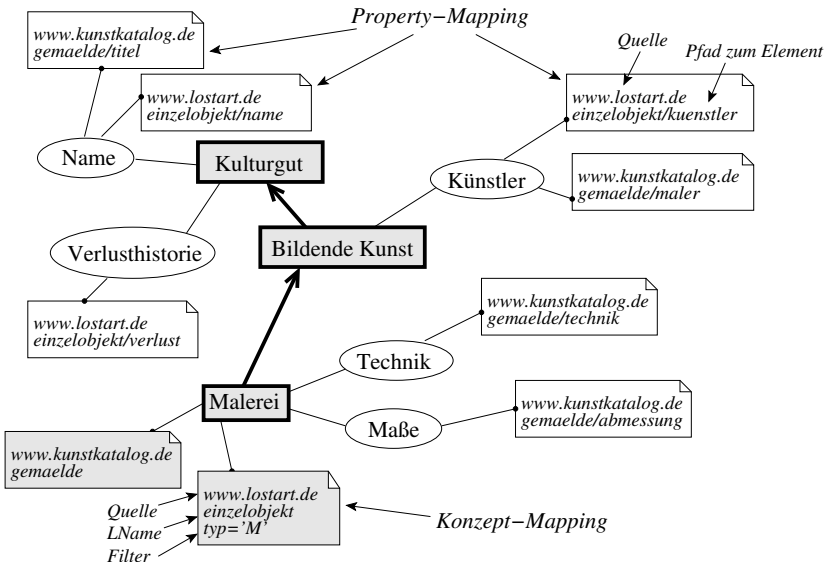


Abbildung 3: Abbildungsinformationen für die Konzepthierarchie

Im weiteren Verlauf dieser Arbeit wird auf die verwendeten Konzept- und Property-Mapping-Vorschriften nicht weiter eingegangen. Sie bilden einen wesentlichen Teil des Mediatormodells, sind aber für den zu entwickelnden Cache nicht relevant. Wie bei der genaueren Betrachtung der Cache-Verwaltung in Abschnitt 4.2 deutlich wird, sind die Cache-Einträge entweder für jedes Konzept oder aber für jede Quelle getrennt zu verwalten. Bei beiden Varianten geht man davon aus, dass die bezüglich den ausgeführten Anfragen gespeicherten Ergebnisse dem Schema der Anfrage entsprechen. Transformationen unter Verwendung der Abbildungsinformationen finden somit immer unabhängig vom Cache statt, entweder bevor die Ergebnisse in den Cache geschrieben werden oder nachdem sie erneut gelesen wurden.

Anfrageformulierung und -bearbeitung

Die Anfrageformulierung im YACOB-Mediator erfolgt mittels einer CQuery genannten Erweiterung von XQuery. Syntaktisch folgt CQuery der Notation von XQuery, die Unterschiede sind vor allem semantischer Natur. CQuery bietet Operationen, die sowohl auf Konzept- als auch auf Datenebene anwendbar sind und darüber hinaus einen Übergang zwischen beiden Ebenen ermöglichen. Eine typische CQuery-Anfrage sieht wie folgt aus:

```
FOR $c IN concept[name='Malerei']
LET $e := extension($c)
WHERE $e/kuenstler = 'van Gogh'
RETURN
  <bild>
    <titel>$e/titel</titel>
    <kuenstler>$e/kuenstler</kuenstler>
  </bild>
```

In der **FOR**-Klausel werden die relevanten Konzepte ausgewählt. Der Übergang von Konzept- zur Instanzebene wird in der **LET**-Klausel mittels der vordefinierten Funktion extension vollzogen. Auf die somit bestimmten Instanzen wird dann das Filterprädikat der **WHERE**-Klausel angewandt. Die Operationen der **FOR**-Klausel werden ausschließlich auf den Metadaten, also im Mediator ausgeführt. Die Auswertung der extension-Funktion sowie der Filterbedingung sind im Gegensatz dazu mit Zugriffen auf Quellsysteme verbunden. Die **RETURN**-Klausel hat die gleiche Bedeutung wie in XQuery: das Anfrageergebnis wird entsprechend der vorgegebenen XML-Struktur ausgegeben.

Wichtig für den entwickelten Cache ist, dass eine globale CQuery-Anfrage in mehrere XPath-Teilanfragen umgeformt wird, die von den Wrappern der Quellen beantwortet werden können. Im Filterprädikat der **WHERE**-Klausel werden Teile der zuvor bestimmten Menge von Instanzen mittels XPath-Pfadausdrücken gefiltert. Hier können die üblichen Vergleichsoperatoren, wie z.B. ’$<$’, ’$>$’ oder ’$=$’, angewandt werden. Die filterbaren Attribute der Objekte sind im YACOB-System alle auf Wertebereichen von Zeichenketten definiert. Operatoren wie ’$<$’ und ’$>$’ werden unterstützt, obwohl sie in Verbindung mit Zeichenketten und dem Anwendungskontext des Mediators auf den ersten Blick eventuell nicht sinnvoll erscheinen. Stellt man sich aber z.B. eine Anfrage der Form „alle Werke von Künstlern mit Anfangsbuchstaben des Nachnamens von A bis K“ vor, so ist ein möglicher Nutzen dieser Operatoren, auch wenn eine solche Suchanfrage sicher selten vorkommt, durchaus erkennbar. Zusätzlich zu den üblichen Operatoren wird ein spezieller Ähnlichkeitsoperator für Zeichenketten ’$\sim=$’ von CQuery unterstützt. Dieser Operator ermöglicht eine unscharfe Suche entsprechend der angegebenen Werte. Ohne einen solchen Operator wäre eine erfolgreiche Suche nur möglich, wenn man zumindest einige der möglichen Attributbelegungen exakt im Voraus angeben kann. Näheres zur Umsetzung des Textähnlichkeitsoperators und zu resultierenden Problemen für die Anfragebearbeitung im Cache wird in Abschnitt 5.4 behandelt.

Die Semantik in CQuery formulierter Anfragen lässt sich leicht mit Hilfe von Algebraoperationen beschreiben. Die Umformung erfolgt mit Hilfe einfacher Regeln. Zu dem oben dargestellten CQuery-Ausdruck ergibt sich unter Anwendung dieser Regeln folgender vereinfachter Algebraausdruck:

$$\biguplus_{c \in CExpr} IExpr(c)$$

CExpr korrespondiert zur **FOR**-Klausel und *IExpr* zur **WHERE**-Klausel. Der Ausdruck *CExpr* bestimmt die Konzeptmenge. Hier finden Operationen zur Auswahl von Konzepten, wie z.B. Filterangaben, Traversierung oder Mengenoperationen, Anwendung. *IExpr* wird auf Instanzebene ausgewertet, d.h. auf jeder durch *CExpr* bestimmten Konzept-Extension. Die Ergebnisse der Anwendung von $IExpr(c)$ auf jedes c werden durch den Vereinigungsoperator $\uplus$ zu einem Gesamtergebnis kombiniert. Weitere Operatoren auf Instanzebene, die mehrere Extensionen umfassen (wie z.B. Verbunde), werden erst später angewandt. Sie sind somit nicht relevant für das vorgestellte Caching-Konzept.

Die Anfragesprache CQuery und der Ablauf der Anfragebearbeitung werden detailliert in [SGHS03] und [SGS03] erläutert. Der gesamte Ablauf ist in Abbildung 4 dargestellt ([KSGH03]). Der Algorithmus verarbeitet nur elementare Anfgrageausdrücke über Konzeptmengen und deren Extensionen.

Gegeben:
Anfrageausdruck der Form $\biguplus_{c \in CExpr} IExpr(c)$
Ergebnis $R := \{\}$

```
1  Berechne Konzeptmenge C := CExpr
2  forall c ∈ C do
3      /* Anfrage nach XPath übersetzen */
4      q :=toXPath(IExpr(c))
5      /* Anfrage q im Cache suchen → Ergebnis ist R_c,
6      q̄ wird als Komplementäranfrage zu q zurückgegeben */
7      R_c := cache-lookup(q, q̄)
8      if R_c ≠ {} then
9          /* Cache-Eintrag gefunden */
10         R := R ⊎ R_c
11         q := q̄
12     fi
13     if q ≠ empty then
14         q_s :=translate-for-source(q, CM(c))
15         R_s :=process-source-query(q_s, s)
16         R := R ⊎ R_s
17     fi
18 od
```

Abbildung 4: Ablauf der Anfragebearbeitung

Die Auswertung der Operationen erfolgt auf Konzeptebene. Die Anfrage wird zunächst nach XPath übersetzt (Zeile 4) und dann auf die zuvor bestimmten Extensionen angewandt. Zu jedem der ermittelten Konzepte wird dann versucht, den extensionsbezogenen Teil *IExpr* aus dem Cache zu beantworten (Zeile 7). Werden entsprechende Ergebnisdaten im Cache gefunden kann auf eine Quellanfrage verzichtet werden. Die Prozedur *cache-lookup* bestimmt das im Cache vorliegende (Teil-)Ergebnis (eventuell durch Anwendung weiterer Filter auf die gespeicherten Daten) zu einer Anfrage q und eine resultierende Komplementäranfrage in $\overline{q}$, die bei Ausführung auf den Quellsystemen den nicht gespeicherten Teil des Ergebnisses liefert.

Sollte eine solche nicht-leere Komplementäranfrage resultieren, wird diese für alle unterstützenden Quellen anhand der Konzept-Mappings $CM(c)$ in entsprechende Anfragen im jeweiligen Quellenformat übersetzt (Zeile 14). Die nach Ausführung dieser Anfragen auf den Quellsystemen erhaltenen Ergebnisdaten werden, wie zuvor die durch den Aufruf von *cache-lookup* erhaltenen Daten, mit der globalen Ergebnismenge vereint (Zeile 16). Liegt im Cache nur ein Teil der Ergebnismenge vor, so kann dieser Teil dem Nutzer auch vor Beantwortung der Komplementäranfrage durch die Quellen zurückgegeben werden. Lesen aus dem physischen Speicher des Caches parallel zur Ausführung der Quellanfragen kann zu einer Beschleunigung der Anfragebeantwortung führen.

Für diese Arbeit wird sich die Einschränkung auf den extensionsbezogenen Teil *IExpr* bei der Suche nach passenden Einträgen im Cache als wesentlich erweisen. Die Filterprädikate der in XPath formulierten Teilanfragen leiten sich aus diesem Ausdruck ab.

Aus dem Aufruf der Prozedur *cache-lookup* für jedes vorher bestimmte Konzept ist zu erkennen, dass der Cache hier unterhalb der Konzeptebene angesetzt ist. Die Einträge im Cache werden für jedes Konzept getrennt verwaltet. Denkbare Alternativen werden in Abschnitt 4.2 diskutiert.

Architektur

Das YACOB-Mediatorsystem ist vollständig in Java implementiert, wobei auf einige Standardtechnologien und frei verfügbare Module zurückgegriffen wurde. So erfolgt die Verarbeitung von XML-Daten mittels JAXP (*Java API for XML Processing*), der Zugriff auf die Quellsysteme bzw. Wrapper über Web Services und die Verwaltung und Manipulation des Konzeptmodells mittels Jena (*Java API for RDF*). Damit die Daten eines jeden Quellsystems in das globale Mediatorschema überführt werden können, werden quellenspezifische XSLT-Regeln generiert, mit deren Hilfe die Umwandlungen erfolgen. Die gesamte Architektur des Systems sowie das Zusammenwirken der einzelnen Komponenten inklusive des Caches ist in Abbildung 5 dargestellt.

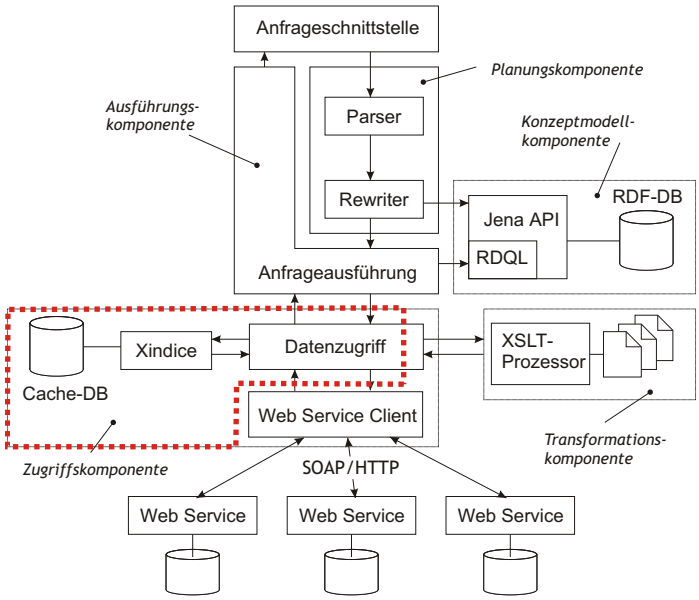


Abbildung 5: Architektur des YACOB-Mediatorsystems

Die Quellen sind über Wrapper verbunden die einfache XPath-Anfragen bearbeiten und die Ergebnisse als XML-Dokumente zurückliefern. Diese Wrapper werden über die Zugriffskomponente mittels SOAP angesprochen. Da die Wrapper als Web Services realisiert sind können sie im Mediator, an den Quellen oder aber auch an einem dritten unabhängigen Ort platziert werden. Die Konzeptmodell-Komponente realisiert auf Basis des Jena-API die Verwaltung der Metadaten, also der Konzepte, ihrer Eigenschaften und Beziehungen sowie der Abbildungsinformationen auf die Quellsysteme, in Form eines RDF Graphen. Die Anfrageplanungs- und Anfrageausführungskomponente bearbeiten Anfragen auf dem globalen Schema.

Mittels der Transformationskomponente werden die nötigen Umwandlungen zwischen dem globalen und den quellenspezifischen Schemata vorgenommen.

Der gesondert hervorgehobene Teil der Zugriffskomponente, der Cache mit physischem Speicher und dem zugehörigen Datenzugriff, stellt den Teil dar, dessen Realisierung Inhalt der vorliegenden Arbeit ist. Die Zugriffskomponente nimmt XPath-Anfragen von der Ausführungskomponente entgegen und versucht, diese aus dem Cache zu beantworten. Ein XPath-Parser wertet die Anfragen aus und im Cache gefundene Teile der Ergebnismenge werden aus dem physischen Speicher, der XML-Datenbank XINDICE, gelesen und kombiniert. Sind die Ergebnisdaten unvollständig, so werden entsprechende Anfragen mittels SOAP an die Wrapper der Quellen weitergeleitet. XML-Daten, die als Ergebnis eine Quellanfrage erhalten werden, werden, nach der Umwandlung in das globale Schema durch Anwendung quellenspezifischer XSLT-Regeln, in der Cache-Datenbank gespeichert.

Vorstellbare Anwender des YACOB-Systems sind vorwiegend Kunsthistoriker, Anwälte, Vertreter von Auktionshäusern und andere Interessierte. Eine direkte Formulierung der Anfragen in CQuery würde diese Personen sicher überfordern. Aus diesem Zweck wurde eine grafische Benutzerschnittstelle entwickelt, in der Informationen über Konzepte, deren Eigenschaften und Kategorien explizit einbezogen sind. Die Oberfläche umfasst die Darstellung der Konzepte, ihrer Eigenschaften sowie die Präsentation der Ergebnisdaten. Der Nutzer kann die Ontologie durchlaufen und dabei gewünschte Konzepte und Kategorien auswählen um das Ergebnis schrittweise zu verfeinern oder zu erweitern. Die angepasste Darstellung des Formulars ermöglicht eine Verfeinerung der Suche durch Einschränkung der definierten Konzepteigenschaften. Die Angabe in den Eingabefeldern ermöglicht dabei eine scharfe und unscharfe Suche, letztere umgesetzt durch einen speziellen von CQuery unterstützten Textähnlichkeitsoperator. Die aktuelle Version der grafischen Schnittstelle ist zu finden unter:

`http://arod.cs.uni-magdeburg.de:8080/Yacob/index.html`.

In diesem Abschnitt wurde ein Überblick über das YACOB-Mediatorsystem gegeben. Besonderes Augenmerk lag dabei auf den Aspekten, die maßgeblichen Einfluss auf den zu realisierenden Cache haben. Für Details wird auf existierende Arbeiten zum YACOB-Mediator verwiesen ([SGHS03, SGS03]).

2.3 Caching in ontologiebasierten Mediatoren

Alle Ansätze für Mediatorsysteme stoßen auf Probleme bezüglich der Antwortzeit für den Anwender und ihrer Skalierbarkeit. Besondere Anforderungen bestehen darin, zum Einen dem Nutzer eine angemessene Antwortzeit auf seine Anfragen zu gewährleisten und zum Anderen die Auslastung der integrierten Quellen durch redundante oder irrelevante Anfragen zu minimieren. Gerade wenn, wie im World Wide Web, Datennetze mit begrenztem Durchsatz integriert werden, rückt die Bedeutung dieser globalen Effizienz weiter in den Vordergrund. Der Zugriff auf die Quellsystemne über Wrapper, der eventuell mit Transformationen verbunden ist, ist im Vergleich zum Zugriff auf ein lokales Datenbanksystem wesentlich ineffizienter und führt zu langen Wartezeiten.

Speziell beim Anwendungshintergrund des YACOB-Mediators ist zudem die Möglichkeit einer interaktiven Anfrageverfeinerung von großer Bedeutung, die auf dem zu erwartenden Verhalten der Anwender des Systems beruht: Um zunächst einen Überblick über vorhandene Objekte zu erhalten, wird anfangs eine relativ ungenaue Anfrage gestellt. Im weiteren Verlauf wird das erhaltene Ergebnis verfeinert, indem die Ergbenismenge durch zusätzliche Konjunktionen eingeschränkt oder durch Angabe weiterer Disjunktionen erweitert wird. Gerade hier kann das Zwischenspeichern zuvor erhaltener Ergebnisse in einem Cache von großem Nutzen sein. Die gespeicherten Ergebnisse können in folgenden Schritten dem Nutzer ohne Zugriff auf die Quellsysteme und somit in angemessener Zeit präsentiert werden. Werden zudem nur die Daten bei den Quellsystemen angefragt, die noch nicht im Cache vorliegen, führt dies zu einer Auslastung der Quellsysteme.

Die entscheidenden Fragen hinsichtlich des Web-Caching sind, neben der bereits geklärten Frage nach dem *Warum*, folgende:

- *Wo* sollen Datenobjekte repliziert werden? (Auf welchen Rechnern?)
- *Was* soll repliziert werden? Hierzu zählt auch die Frage nach einer geeigneten Verdrängungsstrategie bei begrenztem Platz im Cache.
- *Wann* sollen Datenobjekte geladen werden? Dieser Punkt betrifft eher Techniken des Vorladens, die aber bei Anwendung automatischer Algorithmen zur Auswahl der Daten und der Replikationsrechner gleitend in Techniken des Caching übergehen.
- *Wie* sollen Daten auf ihre Aktualität getestet werden? Diese Frage beschäftigt sich mit der Cache-Kohärenz.

Die Antworten auf die einzelnen Fragen werden für den Anwendungskontext dieser Arbeit implizit mit der Beschreibung des Caches gegeben. Eine Zusammenfassung der wichtigsten Punkte findet sich in Abschnitt 4.1. Antworten im Bezug auf das Web-Caching allgemein und weiterführende Aspekte zur Problematik finden sich in [Wei02].

Frühere Caching-Ansätze, wie das *Tupel-* oder *Page-Caching*, erweisen sich für die geschilderten Zwecke als problematisch oder gar unbrauchbar. *Page-Caching* ist weit verbreitet in Client-Server-Systemen. Dabei geht man davon aus, dass jede Anfrage auf das Level der von ihr angefragten Seiten geteilt werden kann, d.h. dass jeder Teil genau einer angefragten Seite entspricht. Bei einem Cache-Fehler wird eine Anfrage für jede Seite, die nicht im Cache gefunden wurde, an den Server gesendet. Dieses Vorgehen ist für Web-basierte Datenintegrationssysteme meist unbrauchbar, da die Suche über Schlüsselwörter oft die einzige Anfragemöglichkeit darstellt und die Architektur der integrierten Quellen generell verborgen ist. Beim *Tupel-Caching* werden einzelne Tupel im Cache gespeichert, was eine höhere Flexibilität als bei der Speicherung ganzer Seiten ermöglicht, aber auch höheren Verwaltungsaufwand mit sich bringt. Vor allem viele relativ kleine Anfragen können die fuer eine Suche im Cache benötigte Zeit merklich erhöhen. Neue Anfragen müssen immer gegen alle im Cache gespeicherten Tupel einzeln geprüft werden.

Hauptnachteil beider Vefahren ist das Fehlen semantischer Daten über die gespeicherten Daten. Es ist nicht möglich zu entscheiden, ob das Ergebnis einer Anfrage komplett im Cache vorliegt oder nur teilweise. Neue globale Anfragen müssen immer auch in die Formate der Quellen transformiert und weitergeleitet werden um entscheiden zu können, ob das gespeicherte Ergebnis vollständig ist oder nicht. Gerade im Anbetracht der Bedeutung der interaktiven Anfrageverfeinerung führt dies nicht zur gewünschten Auslastung der Quellsysteme und zur Beschleunigung der Anfragebearbeitung.

Im Gegensatz dazu hat sich das semantische Caching in vergleichbaren Szenarien als geeignetes Mittel zur Erfüllung der genannten Anforderungen erwiesen. Beim semantischen Caching werden die gespeicherten Daten zusammen mit Daten über die Daten gespeichert. Im Bezug auf die betrachteten Mediatorsysteme sind das erhaltene Ergebnisdaten und die zugehörigen ausgeführten Anfragen, die eine semantische Beschreibung dieser Ergebnisse darstellen. Anhand dieser Beschreibung der im Cache liegenden Daten wird dann bei Eintreffen einer neuen Anfrage entschieden, ob diese ganz oder teilweise aus dem Cache beantwortet werden kann. Dabei werden Ergebnisse aus dem Cache gesammelt und eine entsprechende Komplementäranfrage gebildet, welche bei Übermittlung an die Quellsysteme nur die Ergebnisdaten liefert, die nicht schon aus dem Cache gelesen werden konnten. Offensichtlich führt dies gerade bei der schrittweisen Verfeinerung der Resultate zu positiven Efffekten: Wird die Ergebnismenge erweitert, z.B. durch Angabe zusätzlicher Disjunktionen ('$\vee$') oder Entfernen von Konjunktionen ('$\wedge$'), liegt ein Teil der Daten schon vor und kann zurückgegeben werden während nur nur der verbleibende Teil bei den Quellsystemen anzufragen ist. Bei Einschränkungen der Ergebnismenge, z.B. durch Angabe zusätzlicher Konjunktionen oder Entfernen von Disjunktionen, kann das gesamte Ergebnis aus dem Cache gelesen werden ohne jede Kommunikation mit den Quellen nötig zu machen.

Die Hauptaspekte für die Realisierung eines Caches in Mediatorsystemen, speziell für den YACOB-Mediator, sind folgende:

(1) Integration in die globale Anfragebearbeitung

(2) Anbindung an das verwendete Konzeptmodell

(3) Physische Speicherung der Anfragen und Ergebnisse

(4) Cache-Management-Strategie

Punkt (1) wurde in Abschnitt 2.2 bei der Beschreibung der Anfragebearbeitung im YACOB-System schon betrachtet. Dort wurde die Eingliederung des Caches in die globale Anfrageverarbeitung beschrieben. Vertiefende Aspekte und die Anfragebearbeitung im Cache selbst werden in Kapitel 5 erläutert. Die Definition der Konzepte und ihrer Eigenschaften in einer Ontologie machen Punkt (2), die Anbindung des Mediators an das globale Modell, nötig. In welcher Form das erfolgt wird in Kapitel 4 beschrieben. Folgerungen und Vorteile für Punkt (1) sind wiederum in Kapitel 5 erfasst. Die beiden letzten Aspekte werden bei der Beschreibung der Cache-Verwaltung sowie der Implementierung in den Kapiteln 4 und 6 aufgegriffen und behandelt.

3 Semantisches Caching

3.1 Überblick

Der Name des semantischen Cachings folgt aus dem Bezug zur Semantik der gespeicherten Daten, die in Verwaltung und Anfragebearbeitung einbezogen wird. Beim semantischen Caching werden Daten zusammen mit anderen Daten gespeichert, die die Cache-Inhalte semantisch beschreiben. Die Überprüfung des Cache-Inhaltes auf einen Cache-Treffer oder Cache-Fehler erfolgt anhand dieser beschreibenden Metadaten. Die eigentlichen Daten werden dafür nicht betrachtet.

In Mediatorsystemen sind die ausgeführten Anfragen die semantischen Daten zu den eigentlichen Ergebnisdaten. Ein Eintrag im Cache besteht somit aus der Ergebnismenge, im Fall des YACOB-Mediators einem XML-Dokument, und der zugehörigen Anfrage, beim YACOB-System in Form einer XPath-Anfrage. Trifft eine neue Anfrage ein, so muss anhand der Semantik dieser Anfrage und den im Cache gespeicherten Metadaten entschieden werden, in wie weit sich die Ergebnismengen der beiden Anfragen überlappen. Die Lösung dieser Frage ist allgemein bekannt als das Problem des *Query Containment* ([Ull89, Hal01, LC98]).

Zunächst die formale Definition:

Query Containment:
Eine Anfrage q_1 ist enthalten („contained“) in eine Anfrage q_2, geschrieben $q_1 \subseteq q_2$, wenn für alle Datenbank-Instanzen D das zu q_1 berechnete Ergebnis Teilmenge des Ergebnisses zu q_2 ist, also $q_1(D) \subseteq q_2(D)$.
Die zwei Anfragen sind identisch („equivalent“), wenn $q_1 \subseteq q_2$ und $q_2 \subseteq q_1$.

Ist eine Anfrage q in einer Anfrage c enthalten („contained“), so ergibt sich die zu q passende Antwort durch Anwenden von q auf die zu c gehörige Antwort. Ist c also ein Cache-Eintrag, so liegt die Antwort zu q schon im Cache vor. Wäre andererseits c in q enthalten, so läge ein Teil der Antwort zu q im Cache vor und eine Komplementäranfrage müsste gebildet werden, um die fehlenden Daten zu erhalten.

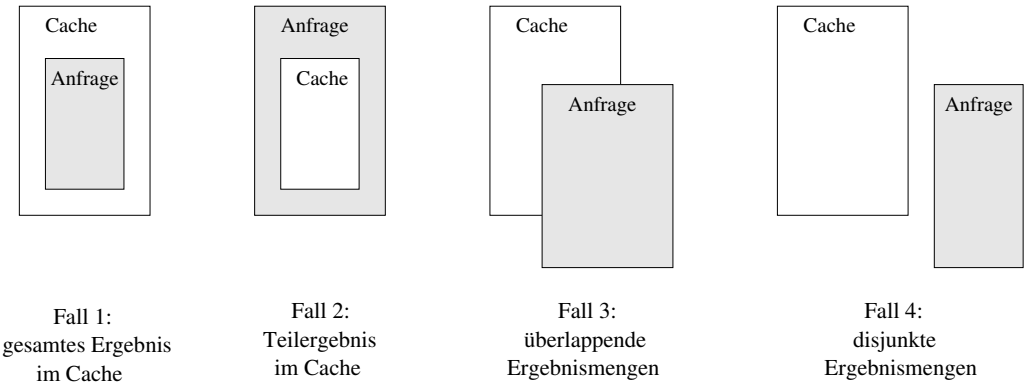


Abbildung 6: Beziehungen zwischen gespeicherter und angefragter Ergebnismenge

In Abbildung 6 werden die Ergebnismengen zweier Anfragen, eine neu ausgeführte und eine im Cache gespeicherte, durch Rechtecke symbolisiert. Die der neuen Anfrage entsprechende ist grau unterlegt, die im Cache gespeicherte weiß. Es lassen sich fünf mögliche Fälle der Beziehung zwischen diesen Ergebnismengen erfassen, von denen vier dargestellt sind.

Der erste der abgebildeten Fälle und der nicht abgebildete Spezialfall der identischen Ergebnismengen sind die einzigen Fälle, in denen die gesamte Ergebnismenge schon im Cache vorliegt. Die Anfrage kann ohne Kommunikation mit den Quellsystemen beantwortet werden. Man spricht von einem „containing match“, bzw. „exact match“ im identischen Fall. Möglichkeit 2 symbolisiert die Situation, dass im Cache nur eine Teilmenge des angefragten Ergebnisses zu finden ist. Der verbleibende Teil muss mit einer komplementären Anfrage von den Quellsystemen angefordert werden. Hier spricht man von einem „contained match“. Bei Fall 3, dem „overlapping match“, überlappen die Ergebnismengen. Die der neuen Anfrage entsprechenden Daten erhält man, indem man die Anfrage an den Cache richtet. Wiederum muss der verbleibende Teil mit einer komplementären Anfrage angefordert werden. Im 4. Fall stehen neue und gespeicherte Anfrage in keiner semantischen Relation. Das im Cache gespeicherte Ergebnis kann nicht zur Beantwortung verwendet werden.

Im allgemeinen unterscheidet man im Fall der überlappenden Ergebnismengen genauer zwischen horizontaler und vertikaler Unvollständigkeit der Daten im Cache. Horizontale Unvollständigkeit liegt vor, wenn die gespeicherten Daten nicht alle angefragten Attribute enthalten. Die fehlenden Attribute müssen zusätzlich angefragt werden, wobei dies nur eindeutig möglich ist, wenn die nötigen Schlüsselattribute ebenfalls projeziert wurden. Im Gegensatz dazu beschreibt vertikale Unvollständigkeit den Umstand, dass Teile der Daten fehlen aber alle Attribute gespeichert sind. Dies tritt zum Beispiel bei selektiven Anfragen ohne Projektionen auf, deren Filterprädikate sich überlappende Ergebnismengen erzeugen. Da im YACOB-Mediator keine Projektionen in den Quellanfragen unterstützt werden und der Cache an die Ontologie mit den definierten Konzepteigenschaften geknüpft wird, ist die horizontale Unvollständigkleit hier nicht weiter zu betrachten. So werden z.B. Verbunde über mehrere Konzeptextensionen immer auf Mediatorebene ausgeführt, der Cache wird davon nicht berührt. Im weiteren Verlauf der Arbeit ist es ausreichend, sich auf die vertikale Unvollständigkeit zu beschränken, die hier in Fall 3 erfasst ist.

Beim semantischen Caching werden die gespeicherten Daten in semantisch zusammenhängende Mengen gruppiert. Diese Mengen werden als semantische Regionen bezeichnet. In Abbildung 6 sind diese semantischen Regionen durch die dargestellten Rechtecke symbolisiert. Die Semantik aller in einer Region gruppierten Daten wird durch die jeweilige Anfrage beschrieben.

Die Verwaltung der entstehenden semantischen Regionen ist ein wesentlicher Faktor eines semantischen Caches. Die im Cache erfassten Daten werden nur anhand der Beschreibung der jeweiligen semantischen Region identifiziert und behandelt. Wesentliche Punkte sind die Frage nach Teilung oder Vereinigung bei Überlappung sowie optimale Größe und Anzahl dieser Regionen. Folgerungen ergeben sich vor allem für die gesamte Cache-Verwaltung und die verwendete Ersetzungsstrategie.

Das Vorgehen zur Suche nach passenden Einträgen während der Anfragebearbeitung ist auch von der Art und Weise der Verwaltung abhängig, ebenso das gesamte Laufzeitverhalten des Caches. Eine detaillierte Analyse dieser Punkte und eine Beschreibung der Verwaltung im realisierten Cache sind in Abschnitt 4.3 zu finden.

3.2 Verwandte Arbeiten

Semantisches Caching ist eine häufig analysierte und diskutierte Thematik. Bei allen beschriebenen Lösungen lassen sich Gemeinsamkeiten erkennen, die aus der Problemstellung unweigerlich folgen. Allerdings folgen aus der Problematik auch einige offene Fragen und Möglichkeiten, deren Betrachtung und Behandlung stellenweise erheblich differiert, was wiederum aus dem jeweils betrachteten Anwendungsfall und seinen Anforderungen folgt. Die verschiedenen Ansätze und ihre Grundlagen werden hier kurz beschrieben.

Das semantische Caching ist eng verknüpft mit der Problematik des *Query Containment*. Beim Caching werden ausgeführte Anfragen mit ihren zugehörigen Ergebnissen gespeichert, die Entscheidung bezüglich Cache-Fehler oder Cache-Treffer erfolgt auf Grundlage der Beschreibung der Ergebnisse, also der zuvor formulierten Anfrage. Lösungen für das *Query Containment* hängen wiederum zusammen mit den Fragen nach der Erfüllbarkeit (*satisfiability*) und der Implikation (*implication*) von Anfragen, entsprechende Literatur zur Thematik findet sich in [Hal01], [LY85], [Ull89] und [GSW96]. Die in diesen Arbeiten vorgestellten Prinzipien und Methoden werden hier angewendet. Die Beschreibung des Algorithmus für Wertebereiche ganzer Zahlen findet sich in Abschnitt 3.4, ein Algorithmus für den speziellen Fall von Wertebereichen auf Zeichenketten wird in Abschnitt 5.1 erläutert.

Der wohl am häufigsten referenzierte Artikel zum semantischen Caching und somit Grundlage für viele Weiterentwicklungen und spezielle Anwendungen ist die Arbeit [DFJ+96], in der erstmals der Begriff semantischer Regionen eingeführt wird. Es wird das ein semantischer Cache für reine Select-Anfragen entwickelt. Damit entspricht dies am besten dem vorliegenden Fall, in dem nur Select-Anfragen ohne Projektionen auftauchen. Die meisten anderen Arbeiten erfassen neben Projektionen oft auch Änderungsoperationen. In [DFJ+96] wird das prinzipielle Vorgehen beim semantischen Caching beschrieben, die Behandlung der semantischen Regionen sowie ein Verdrängungsverfahren basierend auf semantischer Distanz werden vertieft betrachtet.

Ein früherer Ansatz stammt aus [KB96]. Dort erfolgt die Verwendung von Prädikaten ausgeführter Anfragen als Beschreibung der gespeicherten Ergebnisse. Diese Prädikate allerdings werden als Art „black box“ betrachtet, genaue *Containment*-Beziehungen zwischen den Anfragen somit nicht bestimmt. In die Betrachtung fallen aber auch Änderungsoperationen, Projektionen und Verbunde. Um Datenredundanz unter den gespeicherten Tupeln zu vermeiden werden Referenzzähler eingeführt.

Ansätze beider Lösungen werden in [LC98], [LC01] und [LC99] zu Caches speziell für Web-Datenbanken kombiniert. In diesen Arbeiten finden sich unter anderem die semantischen Regionen wieder, werden allerdings *semantic views* genannt.

Zur Realisierung von Überlappungen in den Regionen werden ebenfalls Referenzzähler verwendet. Durch die Überlappungen gibt es im Cache häufig mehrere treffende Anfragen von denen aber keine die korrekte Antwort symbolisiert. Unter diesen muss dann das „best overlapping match“ bzw. „best contained match“ bestimmt werden. Im Gegensatz zur vorliegenden Arbeit ist der Cache aber in den Wrappern realisiert, nicht auf der Seite des Mediators. Beim YACOB-Mediatorsystem wurde sich für einen eng an die Ontologie gekoppelten Cache entschieden. Die Autoren Lee und Chu entwickelten mit ihrem *Find*-Algorithmus allerdings eine interessante Möglichkeit, semantisches Wissen zu verwenden, um aus möglichst vielen Cache-Fehlern Cache-Treffer gewinnen zu können. Andere Ansätze dafür gibt es auch in [ACPS96], dort mittels sogenannter Invarianten. Näheres zur Art und Weise und zu den Möglichkeiten semantisches Wissen derart zu verarbeiten wird in dieser Arbeit kurz im Ausblick (Kapitel 7) skizziert.

Ein recht generelles Framework für semantisches Caching speziell in heterogenen Datenbanken wird in [GG97] vorgestellt, genauere Ausführungen finden sich in [GG99]. Diese Arbeiten stellen eine gute Grundlage zur theoretischen Formulierung semantischer Caching-Modelle dar, denn sie gehen auf alle wesentlichen Punkte des semantischen Cachings ein. Dies umfasst z.B. die Entscheidung, ob Antworten im Cache vorliegen, wie sie gefunden werden, semantische Überlappungen und Unabhängigkeit sowie die Bildung von Komplementäranfragen.

Spezieller Bezug auf heterogene Webquellen wird auch in [CRS99] genommen. Die sich ergebenden besonderen Voraussetzungen und Anforderungen, in Verbindung mit in den Anfragen zugelassenen Projektionen, führen zur Einführung der Begriffe *completeness* und *checkability*. Diese werden genutzt, um die dadurch auftretenden Probleme und eventuellen Komplikationen formal erfassen zu können. Eine Betrachtung dieser Begriffe ist in der vorliegenden Arbeit jedoch nicht nötig, was sich aus der geplanten Anwendung des Caches im YACOB-Mediator ergibt. Welche Voraussetzungen dies tatsächlich unnötig machen wird im Verlauf der Arbeit ersichtlich.

In einigen Arbeiten werden die Vorzüge des semantischen Cachings für mobile Anwendungen entdeckt. So wird in [RD00], [Ren00] und [RD98] das Caching generell genutzt, um die Lücke zwischen Mobilität der Endgeräte und Verfügbarkeit von Informationen zu überbrücken. Ergebnisdaten werden in den Geräten gespeichert und mit aus den ausgeführten Anfragen generierten semantische Daten indiziert. Spezielles Augenmerk liegt auf guten Ersetzungsstrategien, z.B. beruhend auf der augenblicklichen und zu erwartenden Lokalität des Anwenders ([RD00]).

In [Rou91] wird eine inkrementelle Zugriffsmethode für den entwickelten Cache vorgestellt. Die formulierten Ansätze und Prinzipien werden hier kurz am Ende der Betrachtung von Möglichkeiten zur Verwaltung der semantische Regionen (Abschnitt 4.3) aufgegriffen. Dort wird ein Vorschlag zur hierarchischen Verwaltung dieser Regionen und einem abgewandelten inkrementellen Zugriff getätigt, auf den im Verlauf der Arbeit aber nicht tiefer eingegangen wird.

3.3 Einschränkungen im YACOB-Mediatorsystem

Das zuvor beschriebene allgemeine Problem des semantischen Cachings ist bei der Entwicklung eines Caches für das YACOB-Mediatorsystem in einigen Punkten eingeschränkt zu betrachten. Wesentliche Vereinfachungen entstehen durch die Beschränkung auf einfache Selektionsanfragen vor allem bei der Bestimmung der Beziehung zweier Anfragen, dem *Query Containment*.

In Abschnitt 2.2 wurde bei der Beschreibung der Anfragebearbeitung im YACOB-Mediator auch die Eingliederung des Caches beschrieben. Entscheidend für das Caching ist die Einschränkung auf das Filterprädikat der **`WHERE`**-Klausel einer globalen CQuery-Anfrage. Eine globale CQuery-Anfrage wird im Mediator in einzelne in XPath formulierte Teilanfragen transformiert. Die einzelnen Teilanfragen werden durch die Anweisungen in der **`FOR`**- und **`LET`**-Klausel bestimmt, den verbleibenden Teil zur Filterung der Quelldaten bildet das in der **`WHERE`**-Klausel definierte Prädikat. Eine der XPath-Anfragen, die aus der auf Seite 11 abgebildeten CQuery-Anfrage resultieren, hat beispielhaft für eine einzelne Quelle folgende Form:

```
//Malerei[Kuenstler='van Gogh']
```

Die Anfrage filtert alle bezüglich des Konzeptes Malerei gespeicherten Objekte mit der Ausprägung 'van Gogh' im Attribut Künstler und wird in dieser Form im Cache bearbeitet.

Beim semantischen Caching in Mediatorsystemen bilden die ausgeführten Anfragen die semantische Beschreibung der Cache-Inhalte. Im YACOB-Mediatorsystem wird der Cache eng mit der definierten Ontologie verbunden, so dass die Einträge im Cache für die einzelnen Konzepte getrennt verwaltet werden. Bei der Suche nach passenden Einträgen zur bearbeiteten Anfrage werden alle Einträge, die im Kontext zum angefragten Konzept, hier '`Malerei`', gespeichert sind, überprüft. Die Überprüfung bezieht sich dabei auf Überschneidungen bezüglich des angefragten Prädikates, im Beispiel '`[Kuenstler='van Gogh']`', und den Filterprädikaten der Anfragen im Cache. Da man nur die Anfragen zur Beschreibung der Semantik eines Cache-Eintrages nutzt, wird das Problem des *Query Containment* bei der Suche im Cache auf die Prädikate der Anfragen beschränkt.

Die Lösung des *Query Containment* findet aber auch schon außerhalb des Caches auf Ebene der Konzepte statt. Wird z.B. in einer Anfrage q auf die Extension $\mathbf{ext}(c)$ mit zwei Unter-Konzepten c_1 und c_2, wobei gilt $\mathbf{ext}(c) = \mathbf{ext}(c_1) \cup \mathbf{ext}(c_2)$, Bezug genommen, so werden zwei Teilanfragen q_1 (nach $\mathbf{ext}(c_1)$) und q_2 (nach $\mathbf{ext}(c_2)$) gebildet und bearbeitet. Liegt $\mathbf{ext}(c_1)$ nun schon im Cache gespeichert vor, so kann q_1 aus dem Cache beantwortet werden und nur die Beantwortung von q_2 erfordert Kommunikation mit den Quellsystemen. Diese Beziehung zwischen den Konzepten wird während der Anfragebearbeitung außerhalb des Caches bestimmt und ist für die Lösung des *Query Containment* im Cache nicht relevant.

Die formulierten Prädikate werden zur Bearbeitung und zur Speicherung im Cache immer in ihre disjunktive Normalform transformiert. Die Verfahren der Anfragebearbeitung und Cache-Verwaltung basieren alle auf Ausdrücken bestimmter Form.

Die Ausdrücke dürfen nur Teilziele enthalten, die untereinander alle durch ein logisches „Und“ (’$\wedge$’) verknüpft sind. Im weiteren Verlauf der Arbeit werden Ausdrücke dieser Form *konjunktive Ausdrücke* genannt. Disjunktiv, also mit logischem „Oder“ (’$\vee$’), verknüpfte Teile erhalten jeweils einen eigenen Cache-Eintrag (in den im weiteren Verlauf angegebenen Beispielanfragen wird auch „and“/„or“ statt „Und“/„Oder“ verwendet). Die getrennte Behandlung der einzelnen Konjunktionen ist Bedingung für das umgesetzte Suchverfahren im Cache und vereinfacht die Cache-Verwaltung, wie in Kapitel 4 gezeigt wird.

Im weiteren Verlauf der vorliegenden Arbeit werden die Prädikate vereinfachend als Anfragen bezeichnet. Da die im Cache bearbeiteten Anfragen nur aus Angabe des selektierten Konzeptes und einem Filterprädikat bestehen und die Cache-Einträge den jeweiligen Konzepten zugeordnet werden, ist eine formale Reduzierung der betrachteten Anfrage auf das Prädikat ohne Probleme möglich.

Das YACOB-Mediatorsystem wurde zur Suche nach kulturhistorischen Gütern in verschiedenen Datenquellen im WWW entwickelt. Die Suche umfasst dabei für den Anwender nur einfache Selektions-Anfragen ohne Projektionen oder Verbunde. Operationen dieser Art auf Ebene der Konzept-Extensionen bleiben dem Nutzer verborgen. Die Anbindung des Caches an das Konzeptmodell und die beschriebene Einschränkung auf die Filterprädikate vereinfacht das Vorgehen bei der Anfragebearbeitung und Cache-Verwaltung, unterstützt durch das Fehlen jeglicher Änderungsoperationen.

In anderen Beiträgen diskutierte Aspekte wie *checkability*, *derivability* oder *completeness*, erwähnt in Abschnitt 3.2, können somit ebenfalls aus den Überlegungen ausgeschlossen werden.

Die zur Einschränkung der Ergebnisse formulierten Filterprädikate haben dem Suchcharakter des Systems folgend eine einfache Form. Die Benutzerschnittstelle ermöglicht die Auswahl der Konzepte und die Angabe von Filterbedingungen über entsprechende Eingabefelder. Die gebildeten Prädikate enthalten den Eingaben entsprechende Bedingungen der Form $A\ \theta\ c$, A ist der Bezeichner eines Attributes (einer Eigenschaft) und c eine Konstante. Die Menge möglicher Operationen Θ, $\theta \in \Theta$, bestimmt sich durch den Wertebereich der Konstante c und die im System vorgesehenen Einschränkungsmöglichkeiten. Dabei werden üblicherweise nur die Standard-Vergleichsoperatoren unterstützt. Im Fall von Wertebereichen ganzer Zahlen könnte z.B. $\Theta = \{<, \leq, =, \geq, >\}$ gelten. Eine Lösung für das *Query Containment* auf ganzzahligen Wertebreichen unter Berücksichtigung genau dieser Operatormenge wird im nächsten Abschnitt vorgestellt. Die einzelnen Teilziele sind dabei alle nur durch logisches „Und“ und/oder logisches „Oder“ verbunden. Die nötige Umformung in die disjunktive Normalform ist somit ohne Umstände möglich.

Zusätzlich zu den genannten Einschränkungen in der Anfrageformulierung hat die Bedeutung der Möglichkeit zur interaktiven Anfrageverfeinerung wesentlichen Einfluss auf das Caching. Gespeicherte Ergebnismengen werden vorwiegend erweitert oder eingeschränkt, um das Suchergebnis zu verfeinern.

Bei der Suche nach passenden Einträgen im Cache wird die Beziehung zwischen gespeicherter und neuer Anfrage durch einen sogenannten *match-type* ausgedrückt.

match-type (q, c)	Situation	(Teil-)Ergebnis im Cache	Komplementär-anfrage
exact	Ergebnismengen zu q und c identisch	Daten zu c	keine
containing	c containing q	q auf c's Daten	keine
contained	c contained in q	Daten zu c	$q \wedge \neg c$
overlapping	Ergebnismengen zu c und q überlappen	q auf c's Daten	$q \wedge \neg c$
disjoint	Ergebnismengen zu c und q disjunkt	nichts	q

Tabelle 1: Beziehungen zwischen neuer Anfrage q und gespeicherter Anfrage c

Die Bestimmung dieses Wertes ist Inhalt der Frage des *Query Containment*. Die in Abschnitt 3.1, Abbildung 6, eingeführten möglichen Beziehungen zwischen den Ergebnismengen sind folgenden, speziell im YACOB-Mediatorsystem zu beachtenden, Situationen zuzuordnen. Dabei ist ebenfalls der jeweils assoziierte *match-type* angegeben.

1. Identische Anfrage → *match-type*: „exact match“
   ⇒ Ergebnis aus Cache verwenden

2. Angabe zusätzlicher konjunktiv verknüpfter Teilziele oder Entfernung disjunktiv verknüpfter Teilziele → Ergebnismenge eingeschränkt
   → *match-type*: „containing match“ *(c is containing q)*
   ⇒ Anfrage an Cache

3. Entfernung konjunktiv verknüpfter Teilziele oder Angabe zusätzlicher disjunktiv verknüpfter Teilziele → Ergebnismenge erweitert
   → *match-type*: „overlapping match“ oder „contained match“ (*c is contained in q*)
   ⇒ Anfrage an Cache und Komplementäranfrage an Quelle

4. Kein Teil im Cache → *match-type*: „disjoint match“
   ⇒ Anfrage an Quelle

Tabelle 1 fasst die geschilderten Situationen noch einmal zusammen.

Die Möglichkeiten sind sortiert nach ihrer Güte aufgelistet. Der bestmögliche Fall ist natürlich ein „exact match“, da man in diesem Fall alle benötigten Daten erhält indem man einfach das entsprechende Dokument aus dem Cache liest. Findet man kein „exact match“, so ist der Fall des „containing match“ der nächst Beste. Auch hier liegen alle benötigten Daten im Cache vor und eine Anfrage an die Quellsysteme ist unnötig. Allerdings müssen die Daten im Cache hier noch entsprechend der neuen Anfrage gefiltert werden (im Cache gespeichert ist eine Obermenge der Ergebnismenge). Existiert kein Eintrag, mit dessen Hilfe man direkt ein Gesamtergebnis erhält, so hat man im Fall von „contained“ und „overlapping match“ die Möglichkeit, zumindest einen Teil des Ergebnisses aus dem Cache zu lesen. In diesen beiden Fällen resultiert eine Komplementäranfrage, die im weiteren Verlauf des *cache-lookup* anstatt der originalen Anfrage zu bearbeiten ist und den nicht bereits gespeicherten Teil anfragt.

Aufbauend auf den formulierten Bedingungen und definierten Beziehungen wird im nächsten Abschnitt eine Lösung für das eingeschränkte Problem des *Query Containment* auf ganzzahligen Wertebreichen entwickelt.

3.4 Query Containment auf Wertebereichen ganzer Zahlen

In den letzten Abschnitten wurde die enge Verknüpfung des semantischen Cachings zum Problem des *Query Containment* deutlich. Um entscheiden zu können, ob das Ergebnis einer zuvor im Cache gespeicherten Anfrage zur Beantwortung einer weiteren Anfrage verwendet werden kann, ist die Beziehung zwischen neuer und gespeicherter Anfrage zu bestimmen. Die Bestimmung dieses sogenannten *match-type* ist Inhalt des *Query Containments*. Aufgrund dieses engen Zusammenhanges wird hier, aufbauend auf der in Abschnitt 3.1 gegebenen formalen Definition und den im letzten Abschnitt erfassten Einschränkungen, der in gängiger Literatur zu findenden Lösung ein eigener Abschnitt gewidmet. Die dem vorgestellten Algorithmus zu Grunde liegenden Annahmen und Voraussetzungen sind jedoch noch immer allgemeinerer Natur als die im YACOB-Mediator auftretenden, weshalb das tatsächliche Vorgehen im speziellen Anwendungsfall später in Abschnitt 5.1 beschrieben wird. Hier soll der prinzipielle Ablauf des implementierten Algorithmus verdeutlicht werden.

Die Lösung des Problems ist Teil vieler bekannter Arbeiten und eng verknüpft mit den Problemen der Erfüllbarkeit (*satisfiability*) und Implikation (*implication*). Für eine Übersicht der geleisteten Beiträge zu diesen Problemstellungen sei auf [GSW96] verwiesen. Entscheidend für die vorgestellte Lösung ist die Beschränkung des allgemeinen Problems des *Query Containment* auf die Prädikate der Anfragen, die einfache Form dieser Prädikate (nur Attribut-Konstante-Verknüpfungen) sowie die Einschränkung auf konjunktive Ausdrücke.

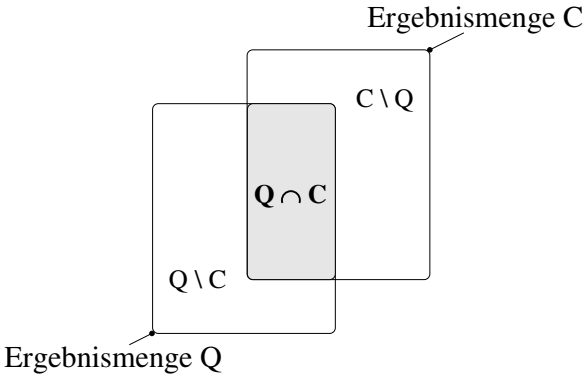


Abbildung 7: Beziehung der Ergebnismengen zweier Anfragen q und c

Die Grundlage für den im Folgenden beschriebenen Algorithmus, der auf den Arbeiten [LY85], [LC01] und [RD98] basiert, ist eine Umsetzung der Lösung des Erfüllbarkeit-Problems. Zwei Anfragen q und c sind gegenseitig disjunkt („disjoint match“), haben also eine leere Schnittmenge in ihren Ergebnissen, wenn $q \wedge c$ unerfüllbar (*unsatisfiable*) ist.

Ebenso gilt folgende einfache Folgerung aus [RD98]:

$$\begin{aligned} (q \rightarrow c) &\Leftrightarrow \neg(q \rightarrow c) = false \\ \neg(q \rightarrow c) &= \neg(\neg q \vee c) \\ \neg(\neg q \vee c) &= q \wedge \neg c \qquad\qquad = false(!) \end{aligned}$$

Gilt $q \rightarrow c$ (q impliziert c), so ist die Antwort zu q ein Teil der der Antwort zu c, eventuell sind beide Mengen auch identisch. Aus der abgebildeten Folgerung ist erkennbar, dass eine Überprüfung auf $q \rightarrow c$ einem Test von $q \wedge \neg c$ auf Unerfüllbarkeit (*unsatisfiability*) entspricht. Abbildung 7 verdeutlicht diesen Zusammenhang anschaulich. Q entspricht der Ergebnismenge zur Anfrage q, C der Ergebnismenge zur Anfrage c.

Das Ergebnis zu q ist komplett in der Ergebnismenge zu c enthalten, also q *contained in* c bzw. $q \rightarrow c$, wenn $Q \setminus C \; = \; Q \wedge \neg C$ einer leeren Ergebnismenge entspricht.

Im Folgenden wird davon ausgegangen, dass alle auftretenden Attribute auf einem Wertebereich ganzer Zahlen definiert sind, der '$\neq$'-Operator wird zunächst ausgeschlossen. An diesem allgemeinen Fall (alle endlichen Wertebereiche sind darauf abbildbar) ist das dem Algorithmus zu Grunde liegende Prinzip am einfachsten verständlich. Weiterhin sind nur Teilbedingungen der Form '$A\ \theta\ c$', A ist ein Attribut und c eine Konstante, zugelassen. Eine weitere Annahme ist, dass die Menge (bzw. die Anzahl) der Attribute von vornherein bekannt ist. Diese Annahme ist nicht zwingend und der Algorithmus ist leicht modifiziert auch dann noch anzuwenden, wenn dies nicht der Fall ist. Tatsächlich findet sich in der Implementierung (siehe Kapitel 6) dann auch ein Algorithmus, der gerade diese Annahme und einige weitere nicht ausnutzt. Dadurch wird das Vorgehen allerdings etwas komplizierter, weshalb die Einschränkungen an dieser Stelle zunächst in Kraft treten.

Das allgemeine Prinzip ist, alle in der Anfrage formulierten Teilziele zu lesen und die Menge zugelassener Werte zu erfassen. Dabei werden Intervalle (*ranges*) für jedes bekannte Attribut verwaltet und an die gerade gelesene Bedingung angepasst. Die für jede Anfrage gebildetetn Intervalle werden dann gegeneinander geprüft und aus den gefundenen Überschneidungen wird der *match-type* abgeleitet und an den Aufrufer zurückgegeben.

Der Ablauf des Algorithmus ist in Abbildung 8 auf Seite 27 dargestellt und wird nun im Detail erläutert. Die innerhalb aufgerufenen Prozedur *build-ranges* ist in Abbildung 9 auf Seite 28 dargestellt. Um den Zusammenhang besser zu verdeutlichen, wurden die Zeilen der beiden Prozeduren nicht getrennt nummeriert, sondern bilden eine zusammenhängende Folge. Ein einfaches Beispiel soll die folgenden Schilderungen veranschaulichen. Betrachten wir zwei Anfragen q_1 und q_2:

q_1: `//Grafik[Datum≤1800]`
q_2: `//Grafik[Datum≥1700 and Datum≤1900]`

Angenommen, die Anfrage q_1 wurde bereits ausgeführt und das ihr entsprechende Ergebnis wurde als ein Eintrag im Cache erfasst. Dies sei zudem der einzige Eintrag, der bezüglich des Konzeptes Grafik im Cache gespeichert ist.

Gegeben:
Query q
CachedQuery c
Rückgabe:
match-type zwischen q und c

```
1     exact = true;
2     rq = build-ranges(q);
3     if rq = NULL then return "q unsatisfiable";
4     rc = build-ranges(c);
5     if rc = NULL then return "c unsatisfiable";
6     for i = 1 to #attr do
          /* sei rx_i = [x_i^min, x_i^max] */
7         if q_i^min > c_i^max or c_i^min > q_i^max then return "disjoint";
8         if q_i^min = c_i^min and q_i^max = c_i^max then continue;
9         else exact = false;
10    od
11    if exact = true then return "exact";
12    if match-type(q, ¬c) = "disjoint" then return "containing";
13    if match-type(c, ¬q) = "disjoint" then return "contained";
14    return "overlapping";
```

Abbildung 8: Prozedur *match-type* für ganze Zahlen

Die neue Anfrage q_2 wird nun im Lauf der Anfragebearbeitung an die Cache-Verwaltung übergeben und das im Cache vorliegende Teilergebnis soll durch Aufruf der Prozedur *build-ranges* bestimmt werden.

Zunächst werden die jeder Anfrage entsprechenden *ranges* mittels *build-ranges* gebildet (Zeilen 2 und 4). Diese *ranges* umfassen den gültigen Bereich für jedes mögliche Attribut, dargestellt durch $[min, max]$. Die Intervalle werden als zu beiden Seiten abgeschlossene Intervalle verstanden, min und max stellen also noch zulässige Werte dar. Für den Wert x_i eines Attributes X_i gilt somit $min \leq x_i \geq max$ (mit Ausnahme der gesondert zu behandelnden Intitialisierungs-Werte $[-\infty, +\infty]$, die nicht wirklich angenommen werden können). Bei den hier betrachteten ganzzahligen Wertebereichen wäre eine Realisierung mit offenen Intervallen ebenfalls möglich, denn der Vorgänger und der Nachfolger einer Konstante c sind problemlos bestimmbar. Bei Unterstützung anderer Wertebereiche mit entsprechend definierter Ordnung der Elemente, wie z.B. Wertebereiche auf Zeichenketten, ist dies nicht ohne weiteres möglich (siehe dazu Abschnitt 5.1). Im Ablauf der Prozedur *build-ranges* wird jede Bedingung '$X_i\ \theta\ c$' der betrachteten Anfrage untersucht (Zeile 16) und das für X_i gültige Intervall entsprechend der Konstante c und der Operation θ aktualisiert (Zeilen 18 bis 24). Wird ein Attribut überhaupt nicht eingeschränkt, so entspricht das zugehörige Intervall am Ende der Prozedur, wie zu Beginn initialisiert (Zeile 15), $[-\infty, +\infty]$.

In der *build-ranges*-Prozedur ist ein Test auf Erfüllbarkeit integriert. Findet man ein Attribut, für dessen Intervall $[min, max] : min > max$ gilt, so ist die betrachtete Anfrage unerfüllbar (*unsatisfiable*) und NULL wird zurückgegeben (Zeile 26).

Gegeben:
Query q
Rückgabe:
ranges r, NULL wenn q unsatisfiable

```
15      for i = 1 to #attr do r_i = [-∞, +∞] od
16      forall condition 'X_i θ c' of q do
17          switch (θ) do
                /* sei r_i = [a_i, b_i] */
18              case '=': if c < a_i or c > b_i then return NULL;
19                        r_i = [c, c];
20                        break;
21              case '>': r_i = [max(a_i, c + 1), b_i]; break;
22              case '≥': r_i = [max(a_i, c), b_i];     break;
23              case '<': r_i = [a_i, min(b_i, c - 1)]; break;
24              case '≤': r_i = [a_i, min(b_i, c)];     break;
25          od
26          if a_i > b_i then return NULL;          /* neue Werte!! */
27      od
28      return r;
```

Abbildung 9: Prozedur *build-ranges* für ganze Zahlen

Auf diese Weise wird für alle Bedingungen mit Ausnahme des Gleichheitsoperators geprüft, ob sie in Verbindung mit den zuvor verarbeiteten Teilbedingungen die Erfüllbarkeit der Anfrage verletzen. Da in den entsprechenden Zeilen 21 bis 24 immer nur eine Seite des Intervalls angepasst wird, ist eine Überprüfung *nach* dieser Modifikation (Zeile 26) ausreichend. Teilbedingungen, die ein '=' enthalten, müssen hingegen *vor* Modifikation der Intervalle geprüft werden (Zeile 18), da in diesem Fall beide Seiten angepasst werden und eine nachträgliche Überprüfung somit nicht möglich ist.

Die Anfragen aus dem Beispiel sind beide erfüllbar. Die bestimmten Intervalle beider Anfragen bezüglich des Attributes Datum sind:

$$q_1 : ranges_{Datum} = [-\infty, 1800]$$
$$q_2 : ranges_{Datum} = [1700, 1900]$$

Es ist sofort zu erkennen, dass die Intervalle der beiden Anfragen nicht identisch sind, sich aber auch nicht ausschließen. Vielmehr überlappen die Intervalle. Die Ergebnismenge der Anfrage q_1 umfasst alle Objekte zum Konzept Grafik bis zum Datum 1800, welche somit einen Teil aller Objekte mit Datum zwischen 1700 und 1900 enthält. Der Teil der von q_2 angefragten Ergebnismenge, dessen Elemente im Attribut Datum einen Wert zwischen 1801 und 1900 enthalten, liegt nicht im Cache vor. In der Prozedur *match-type* wird diese Überlappung erkannt und „overlapping match“ wird zurückgegeben. Die Bestimmung dieser Beziehung erfolgt wie geschildert durch verschiedene Kombinationen der untersuchten Anfragen untereinander und Kontrolle auf Erfüllbarkeit der resultierenden Anfragen.

Im Beispiel werden folgende Anfragen gebildet und überprüft:

$q \wedge \neg c \quad \hat{=} \quad q_2 \wedge \neg q_1 =: q'$:

```
//Grafik[Datum≥1700 and Datum≤1900 and ¬(Datum≤1800)]
≙ //Grafik[Datum≥1700 and Datum≤1900 and Datum>1800]
≙ //Grafik[Datum>1800 and Datum≤1900]
```

ist erfüllbar (also $Q \setminus C \neq \emptyset$) und

$c \wedge \neg q \quad \hat{=} \quad q_1 \wedge \neg q_2 =: q''$:

```
//Grafik[Datum≤1800 and ¬(Datum≥1700 and Datum≤1900)]
≙ //Grafik[(Datum≤1800 and Datum<1700) or
            (Datum≤1800 and Datum>1900)]
≙ //Grafik[Datum<1700 or (Datum≤1800 and Datum>1900)]
```

wobei

```
//Grafik[Datum<1700]
```

erfüllbar ist, somit also auch q'' erfüllbar ist ($C \setminus Q \neq \emptyset$). Der Algorithmus liefert beim rekursiven Aufruf in Zeile 12 und 13 also jeweils nicht „disjoint match“ und gibt abschließend „overlapping match“ zurück.

Es fällt auf, dass das angegebene Verfahren effizienter gestaltet werden kann, da innerhalb des *match-type*-Algorithmus dieser zweimal rekursiv aufgerufen wird. Das Prinzip der umgesetzten Lösung soll hier möglichst einfach ersichtlich werden. Eine offensichtlich erstrebliche Variante ist es, den *match-type* in nur einem Lauf über die Intervalle der Attribute zu bestimmen. Eine solche Variante des Algorithmus ist durchaus möglich und wurde im Cache auch umgesetzt. Die wenigen sich ergebenden Veränderungen sind im Abschnitt 5.1 erfasst. Dort wird der schließlich im Cache verwendete *match-type*-Algorithmus beschrieben, der unter anderem das eben geschilderte Vorgehen umsetzt.

Eine Betrachtung von Verknüpfungen der Form '$A \ \theta \ B$', A und B sind Attribute, ist nicht nötig, da solche Verknüpfungen im YACOB-Mediatorsystem nicht vorgesehen sind. Das allgemeine Problem der Erfüllbarkeit von Anfragen ist, unter Berücksichtigung des Ungleichheitsoperators '$\neq$' auf ganzzahligen Wertebereichen, als NP-vollständig bekannt ([RI80]). In allen bekannten Lösungen folgt diese Komplexität aus der Inbetrachtnahme von Verknüpfungen zwischen Attributen. Die Ansätze zum Lösen des Problems führen dann in der Regel auf graphentheoretische Fragen zurück, z.B. bei der Lösung über Konnektivitätsgraphen ([Ull89]). Eine gute Übersicht über existierende Ansätze und resultierende Komplexitäten ist in [GSW96] zu finden. Als wesentliche Erkenntnis wird dort festgehalten, dass die NP-Vollständigkeit aus der Kombination von möglichen Attributverknüpfungen mit dem '$\neq$'-Operator auf ganzzahligen Wertebereichen resultiert. Der entsprechende Beweis wurde zuvor schon in [RI80] erbracht.

Die NP-Vollständigkeit ist im vorliegenden Fall nicht gegeben, weil keine solche Attributverknüpfungen in den Anfragen vorkommen. Einzig Konstantenselektionen werden unterstützt, was zu einer in der Anzahl der Teilziele einer Anfrage linearen Komplexität führt. Auf einen Beweis wird hier verzichtet. Betrachtet man den Algorithmus in Abbildung 8, so ist diese Abschätzung leicht nachzuvollziehen.

Die Prozedur *build-ranges* läuft nacheinander über die einzelnen Teilziele der beiden Anfragen und grenzt das gültige Intervall ein. Dabei ist für jedes Teilziel ein konstanter Aufwand nötig, denn die alten Intervallwerte werden in jedem Schritt nur kontrolliert und gegebenenfalls angepasst. Nach Bestimmung der Intervalle wird über die Anzahl aller möglichen Attribute iteriert und die beiden Intervalle können wiederum mit konstantem Aufwand gegenseitig überprüft werden. Die Anzahl der Attribute ist als Konstante im Voraus bekannt, die obere Grenze für die Komplexität wird somit von der Anzahl der Teilziele in den Anfragen bestimmt.

4 Struktur und Verwaltung des Caches

In diesem Kapitel werden die bisher geschilderten Aspekte und theoretischen Hintergründe des semantischen Cachings auf einen konkreten Anwendungsfall aus der Praxis projiziert. Es wird das Konzept eines semantischen Caches für den in Kapitel 2 vorgestellten YACOB-Mediator entwickelt.

Im nächsten Abschnitt werden zunächst die speziellen Anforderungen, die sich aus Aufbau und Funktion des Mediators ergeben, zusammengefasst. Die im bisherigen Verlauf erarbeiteten Einschränkungen und Voraussetzungen werden ergänzt und die sich ergebenden Anforderungen für die Implementierung abgeleitet. In weiteren Abschnitten wird auf besonders bedeutende Aspekte der Cache-Verwaltung, wie z.B. die Behandlung der entstehenden semantischen Regionen oder die Anbindung an die Ontologie des Mediators, eingegangen. Der gesamte Ablauf der Anfragebearbeitung inklusive der verwendeten Algorithmen, die Implementierung des Caches und die Experimente zur Bewertung werden in separaten Kapiteln behandelt.

4.1 Anforderungen und Voraussetzungen

Ziel der Arbeit ist die Entwicklung eines semantischen Caches für das YACOB-Mediatorsystem. Die in den vorangegangenen Abschnitten erarbeiteten Voraussetzungen, die aus diesem speziellen Anwendungsfall resultieren, werden in der folgenden Übersicht zusammengefasst. Gleichzeitig werden die aus den aufgelisteten Bedingungen resultierenden Anforderungen an die Realisierung des Caches erfasst. Die Auflistung gibt gleichzeitig Antwort auf die in Abschnitt 2.3 formulierten allgemeinen Fragen.

- **Ziel:**

  Realisierung eines semantischen Caches für das YACOB-Mediatorsystem zur Beschleunigung der Anfragebearbeitung und Reduzierung der Quellenauslastung. Der Cache soll Teil des Mediators und unabhängig von den Quellsystemen und Clients sein, womit die Frage nach dem *Wo* geklärt ist.

- **Aufgabe:**

  Zwischenspeichern ausgeführter Anfragen und erhaltener Ergebnisse. (Teilweise) Beantwortung neuer Anfragen durch Suche nach dem maximalen im Cache gespeicherten Teil der Ergebnismenge.

  Da nur bereits ausgeführte Anfragen gespeichert werden und keine Vorlade-Technik Anwendung findet, ist hiermit die Frage nach dem *Wann* beantwortet.

- **Im Cache zu erfassen:**

  XML-Ergebnisdokumente mit zugehörigen XPath-Anfragen:

  – Die Daten in den XML-Dokumenten entsprechen den in den verschiedenen Quellsystemen gespeicherten Daten der angefragten Objekte. Sie liegen bereits transformiert, d.h. dem globalen Schema des Mediators entsprechend, vor.

  - Die jeweilige XPath-Anfrage bildet die semantische Beschreibung der Daten in den zugehörigen XML-Dokumenten. Die Daten werden dadurch in sogenannte semantische Regionen gruppiert. Die Daten jeder Region bilden jeweils eine separaten Cache-Eintrag.

  Damit ist ebenfalls die Frage nach dem *Was*, mit Ausnahme einer Verdrängungsstrategie, beantwortet.

- **Suche im Cache**:

  - Die Suche nach Treffern im Cache erfolgt nur anhand der semantischen Beschreibung der bearbeiteten Anfrage und den semantischen Beschreibungen der Cache-Einträge.
  - Die Integration in die globale Anfragebearbeitung erfolgt durch Realisierung einer Prozedur *cache-lookup*. Diese Prozedur gibt das im Cache gespeicherte (Teil-)Ergebnis und eine eventuelle Komplementäranfrage zurück.
  - Die Komplementäranfrage beschreibt den nicht im Cache gespeicherten Teil der Ergebnisdaten.

- **Bedeutung der interaktiven Anfrageverfeinerung:**

  Das primäre Ziel des Caches ist die Unterstützung einer effizienten interaktiven Anfrageverfeinerung im Mediatorsystem. Folglich ist vorwiegend mit Anfragen zu rechnen, die bereits im Cache gespeicherte Ergebnismengen einschränken (nur Nachselektion auf Cache-Daten) oder erweitern (Komplementäranfrage und eventuell Nachselektion).

- **Anbindung des Caches an das Konzeptmodell:**

  Die Einträge im Cache sind für die einzelnen Konzepte getrennt zu verwalten.

- ***Query Containment* beschränkt auf Prädikate:**

  - Alle im Cache bearbeiteten Anfragen entsprechen der allgemeinen Form

    ```
    //Konzeptname[Filterprädikat]
    ```

  - Aus der Anbindung an das Konzeptmodell und der einfachen Form der XPath-Anfragen folgt bei Lösung des Problems des *Query Containment* eine Einschränkung auf die Filterprädikate der Anfragen.
  - Die Filterprädikate bestehen aus Teilzielen, die nur durch logisches „Und“ und/oder logisches „Oder“ verbunden sind und die die im Mediatormodell definierten Konzepteigenschaften einschränken.
  - Die einzelnen konjunktiven Ausdrücke der disjunktiven Normalform eines Prädikates sind getrennt zu behandeln. Im Cache werden nur Einträge gespeichert, die durch konjunktive Ausdrücke beschrieben werden.
  - Die Beziehung zwischen zwei Anfragen wird durch eine zu implementierende Prozedur *match-type* bestimmt.

- **Physischer Speicher**

  Eine effiziente Methode zur persistenten Speicherung von XML-Daten und den beschreibenden Daten wid benötigt.

- **Implementierung**

  Der Cache wird, wie das YACOB-Mediatorsystem, in Java implementiert. Dies ermöglicht eine Evaluation des Caches und die Einbindung in das laufende System.

- **Ersetzung und Kohärenz**

  Eine Strategie zur Verdrängung von Cache-Einträgen und Wahrung der Kohärenz der Daten ist zu entwickeln. Der entsprechende Abschnitt 4.5 gibt Antwort auf die letzten offenen Fragen nach dem *Was* und *Wie*.

- **Weitere Anforderungen:**

  – Es sind zusätzlich Methoden zum Einfügen, Entfernen und Ersetzen von Cache-Einträgen zu implementieren. Diese sind auch zur Umsetzung der entwickelten Ersetzungsstrategie zu nutzen.

  – Zur Bewertung des Vorgehens sind statistische Methoden umzusetzen.

Fasst man die wichtigsten Punkte noch einmal zusammen, so kann man die Anforderungen (recht vereinfachend) so formulieren: Im Cache werden ausgeführte Anfragen und zugehörige Ergebnisse gespeichert. Die Anfragen bilden die semantische Beschreibung der gespeicherten Ergebnisdaten, die in Form von XML-Dokumenten vorliegen. Wenn eine neue Anfrage bearbeitet wird, so wird im Cache anhand dieser beschreibenden Daten geprüft, ob die Anfrage ganz oder teilweise mit den im Cache gespeicherten Daten beantwortet werden kann. Ist dies der Fall, so wird das (Teil-)Ergebnis aus dem physischen Speicher des Caches gelesen und an das System zurückgeliefert.

4.2 Caching pro Konzept oder pro Quelle

Eine wichtige Entscheidung bei der Entwicklung des Caches ist die Frage, ob man den Cache auf Konzept- oder auf Instanzebene ansetzt. Die enge Anbindung des Caches an das Konzeptmodell wurde als Voraussetzung für die Umsetzung festgehalten. Die Einträge im Cache sind somit für jedes Konzept getrennt zu verwalten. Die Anbindung des Caches an die Quellsysteme führt zu einer Verwaltung der Einträge getrennt für jede Quelle, die das angefragte Konzept unterstützt. Die Anbindung an die Ontologie des Mediators ist bei diesem Vorgehen ebenfalls gegeben. Die Relevanz der Entscheidung für eine der beiden Vorgehensweisen wird in diesem Abschnitt erläutert.

Bei Platzierung auf Konzeptebene werden die Einträge für jedes Konzept getrennt verwaltet, wie bei Beschreibung der Anfragebearbeitung in Abschnitt 2.2 realisiert. Bei der alternativen Angliederung an die Instanzebene werden die Datensätze getrennt für jedes, das angefragte Konzept unterstützende Quellsystem verwaltet.

Der Unterschied wird zunächst in der Form der gespeicherten Daten ersichtlich. Beim Caching auf Konzeptebene werden Daten gespeichert, die dem globalen Schema entsprechen. Die bearbeitete Anfrage wird erst nach der Behandlung im Cache in die entsprechenden Quellformate übersetzt. Im Falle eines Cache-Treffers werden keine weiteren Transformationen benötigt, da das Ergebnis schon im Format des globalen Schemas vorliegt.

Verwaltet man die Daten für jede Quelle getrennt, so erfolgt das Caching unterhalb der Transformationskomponente des Mediators. Anfragen und zugehörige Ergebnisse werden entsprechend den Quellschemata gespeichert. Umformungen der Daten sind somit immer nötig. Ein Cache-Treffer bringt somit nicht den gewünschten Effekt, diese Transformationen zu umgehen. Stellt man sich z.B. eine Quelle vor, die nur persönliche Daten über Künstler liefert, so müssen die der Anfrage entsprechenden Attribute der Quelldaten immer vom Mediatorsystem gefiltert und mit den Daten der eigentlichen Objekte verbunden werden. Beim Caching auf Instanzebene sind diese Operationen auch bei einem Cache-Treffer nötig und einige Attribute werden gespeichert, obwohl sie eventuell nie wieder angefragt werden. Zudem entspricht eine Anfrage nach der Umwandlung in die Quellenformate mehr einzelnen Teilanfragen als im Format der Konzeptebene. In Verbindung mit der höheren Anzahl von gespeicherten Anfragen (resultierend aus der feineren Granularität) führt dies zu einer Verzögerung in der Anfragebearbeitung, da die Beziehung zwischen zwei Teilanfragen wesentlich öfter bestimmt werden muss.

Vorteil des Cachings auf Instanzebene ist die feinere Granularität der Einträge. Diese ermöglicht zum Einen eine Ersetzungsstrategie, die Bezug auf den spezifischen Charakter eines jeden Quellsystems nehmen kann. Zum Anderen sind temporäre Ausfälle von Quellen oder neu eingegliederte Quellen leichter zu erkennen und entsprechend zu behandeln.

Annahme in beiden Fällen ist, dass Ergebnisdaten und Anfrage im gleichen Format gespeichert werden. Das Ergebnis entspricht immer dem Schema der Anfrage. Daraus folgt, dass im Cache selbst keine Transformationen der Daten nötig sind. Die Bestimmung der Beziehung zwischen zwei Anfragen und die Behandlung der Ergebnisdaten unterscheidet sich in beiden Fällen nicht, so dass die Entscheidung keinen Einfluss auf den Ablauf der Anfragebearbeitung hat. Welche Variante vorzuziehen ist, hängt vom Anwendungsfall ab.

Im YACOB-Mediatorsystem ist der Cache direkt an die Konzeptebene gekoppelt. Wie in Abbildung 4 auf Seite 13 dargestellt ist, erfolgt die Suche im Cache getrennt für jedes Konzept und vor der Umformung der Anfrage in die Quellenformate. Wesentlicher Grund dafür ist die erzielte Beschleunigung der globalen Anfragebearbeitung.

4.3 Verwaltung der semantischen Regionen

Werden die Daten wie geschildert zusammen mit ihren beschreibenden Daten gespeichert, so werden sie in semantisch zusammenhängende Mengen gruppiert, genannt semantische Regionen. Jeder Cache-Eintrag repräsentiert eine solche semantische Region, die durch die zum Eintrag gehörende Anfrage beschrieben wird.

Wie schon in Kapitel 2 bemerkt, gibt es verschiedene Möglichkeiten, die entstehenden semantischen Regionen zu behandeln. Allgemein gilt, dass eine semantische Region einen Eintrag im Cache repräsentiert. Jede Region wird durch eine Formel ausgedrückt, welche aus konjunktiv verknüpften Bedingungen besteht und die in der Region enthaltenen Daten beschreibt. Eine solche semantische Region mit ihren Daten und Metadaten entspricht somit dem verbreiteten Begriff der „materialisierten Sicht“. Es ist eine logische Forderung, dass ein Cache keine Daten redundant speichern sollte. Eine einfache Speicherung aller Anfragen und zugehöriger Ergebnisse kann diese Anforderung augenscheinlich nicht erfüllen, die Verwaltung der semantischen Regionen muss also genauer überlegt sein.

Alternativen ergeben sich, wenn neue Regionen durch Quellanfragen entstehen. Dies ist immer dann der Fall, wenn eine Anfrage nicht komplett durch die im Cache gespeicherten Daten beantwortet werden kann. Die in diesem Fall gebildete Komplementäranfrage liefert die fehlenden Daten von der Quelle, welche dann als neuer Cache-Eintrag, beschrieben durch eine neu entstehende semantische Region, in den Cache integriert werden müssen. Der Entscheidungs-Spielraum umfasst im Wesentlichen das Teilen/Vereinen von Regionen und das Verhalten bei auftretenden Überlappungen. Im Folgenden werden die prinzipiellen Möglichkeiten und ihre Auswirkungen auf Effizienz und Ersetzungsstrategie kurz beschrieben und abschließend noch ein Vorschlag zur hierarchischen Verwaltung der Regionen gegeben, der eine Effizienz-Steigerung bei einer hohen Anzahl von Überlappungen verspricht.

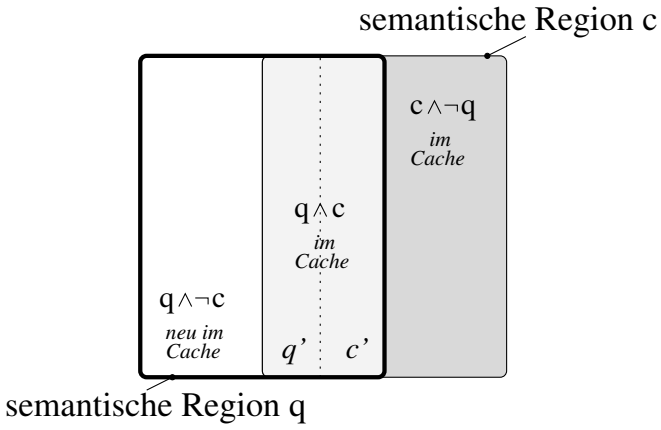


Abbildung 10: Semantische Regionen zu Cache-Eintrag c und Anfrage q

In Abbildung 7 auf Seite 25 ist der Fall eines „overlapping match“ zwischen ausgeführter Anfrage q und gespeicherter Anfrage c verdeutlicht. Abbildung 10 stellt die gleiche Situation ähnlich dar. Die jeweilige Anfrage bildet zugleich die Beschreibung der resultierenden semantischen Region, somit beschreibt c die semantische Region des Cache-Eintrages (grau unterlegt) und q die der neuen Anfrage (dick umrandet). Der überlappende Teil, den beide Regionen gemeinsam enthalten, ist in etwas hellerm Grau dargestellt. Der zweite Fall, in dem eine neue semantische Region entsteht, der Fall des „contained match“, ist analog zu behandeln.

Es resultieren drei (Teil-)Regionen, die auf verschiedene Art und Weise vereinbar sind.

Die semantischen Beschreibungen der drei gegenseitig disjunkten Regionen lauten:

- $c \wedge \neg q$, der Teil der Ergebnismenge C, der kein Bestandteil der Antwort Q ist
- $q \wedge c$, der Teil der Ergebnismenge C, der Bestandteil der Antwort Q ist
- $q \wedge \neg c$, der Teil der Antwort Q, der noch nicht im Cache gespeichert ist und der somit die einzig wirklich „neue“ Region darstellt

Die sich in diesem Fall bietenden Möglichkeiten, die entstehenden Regionen zu behandeln, sind folgende:

1. Alle Regionen zu einer einzigen vereinen.
2. Alle drei Regionen separat speichern und verwalten.
3. c als eine semantische Region belassen ($C \setminus Q$ und $Q \cap C$ zusammenfassen) und $q \wedge \neg c$ als neue Region in den Cache aufnehmen.
4. Region c in die Mengen $C \setminus Q$ und $Q \cap C$ teilen und wiederum $Q \cap C$ mit $Q \setminus C$ vereinen, also q als neue Region integrieren und c zu $c \wedge \neg q$ reduzieren.
5. Ergebnismenge $Q \cap C$ so teilen, dass etwa die Hälfte der zugehörigen Daten zur Menge $Q \setminus C$, der Rest zu $C \setminus Q$ zugeordnet werden kann. Voraussetzung dafür allerdings ist eine besondere Semantik der Anfragen q und c, denn den resultierenden Mengen entsprechend müssen zwei semantische Regionen $(q \wedge \neg c) \vee q' = q \wedge \neg c'$ und $(c \wedge \neg q) \vee c' = c \wedge \neg q'$ gebildet werden können. Das Problem besteht in der Bestimmung von q' und c' zur semantischen Beschreibung der resultierenden Regionen. Vorstellbar ist dies z.B. bei Angabe zulässiger Zahlenbereiche, wo die Mitte ermittelt und die Intervalle leicht angepasst werden können.
6. Im weiteren Verlauf zwei einzelne Regionen behandeln, die sich entsprechend der Anfragen überlappen.

Jede der genannten Strategien ist unter gewissen Aspekten den anderen vorzuziehen. Die wesentlichen Faktoren, die diesbezüglich entscheidenden Einfluss haben, sind vor allem in der zu Grunde liegenden Anwendungsabsicht bzw. in den dadurch gegebenen Annahmen und Voraussetzungen zu finden. Nachdem nun die aus der Anwendung jeder einzelnen Strategie resultierenden Vor- und Nachteile beschrieben werden, sind es auch die im YACOB-Mediator zu findenen praktischen Voraussetzungen, die am Ende zur Begründung der schließlich in der Umsetzung getroffenen Wahl herangezogen werden.

Die erstgenannte Variante, alle drei Regionen zu einer einzigen zu vereinen, scheint vielleicht recht naheliegend, liegen ja im Cache alle benötigten Daten vor. Dadurch wird aber in erster Linie eine viel zu grobe Speicher-Granularität im Cache erreicht. Wird eine über solche Schritte gewachsene Region aus dem Cache entfernt, so wird ein erheblicher Teil des gesamten Cache-Inhaltes gelöscht. Dies kann allerdings nicht Ziel einer guten Cache-Verwaltung sein.

Will man im Gegensatz dazu, wie in der zweiten Variante, alle drei Regionen einzeln speichern, so muss man zum Einen die Daten, die bereits im Cache liegen und bisher einer Region zugeordnet waren ($C \setminus Q$ und $Q \cap C$) trennen und zum Anderen wird die Anzahl der nun sehr kleinen Regionen stark ansteigen, was eine effiziente Anfrageverarbeitung kaum möglich macht. Positiv an diesem Verhalten ist die Möglichkeit, die Zugriffshäufigkeit auf einem sehr feinen Granularitätslevel ausdrücken zu können.

Bleiben die nächsten drei Möglichkeiten, in denen die Regionen jeweils spezieller zusammengefasst werden, als im ersten Fall. Dies führt in einigen Fällen auch zu einer Aufsplittung der „alten“ Region c. Klare Vorteile der einzelnen Varianten sind:

- Das 3. Verfahren ermöglicht eine Ersetzungsstrategie, die Bezug auf den Zeitpunkt der Datenkollektion nehmen kann.

- Im Gegensatz dazu ist man mit dem 4. Vorgehen in der Lage, eine Ersetzungsstrategie umzusetzen, die Bezug auf die Häufigkeit der Anfrage nimmt, z.B. LRU (siehe Abschnitt 4.5). In [DFJ+96] wird diese Methode benutzt um ein Ersetzungsverfahren beruhend auf „semantischer Distanz“ zu realisieren. Eine Verschmelzung erfolgt jedoch nur zwischen Regionen, die den gleichen Kennwert besitzen, welcher sich aus eben genannter „semantischer Distanz“ ableitet.

- Die letztgenannte dieser drei Möglichkeiten scheint recht intuitiv, liegen z.B. numerische Wertebereiche vor. Dann ist sie sehr leicht umzusetzen und ein nutzbarer Effekt wäre beispielhaft für statistische Anwendungen denkbar. Sie ist im vorliegenden Fall allerdings nicht in Betracht zu ziehen.

Bei der Variante der überlappenden Regionen schließlich, wie sie z.B. in [LC98] und [KB96] zu finden ist, werden entstehende Regionen grundsätzlich nie vereint oder geteilt. Jede Anfrage bildet eine neue solche Region. Zwangsläufig müssen sich die Regionen also auch überlappen. Damit trotzdem keine Daten-Redundanz entsteht, wird mit sogenannten *reference counters* gearbeitet, die angeben, wie oft ein Datum im Cache referenziert wird. Hierdurch wird zwar eine einfache Verwaltung und gute Ersetzungsstrategie möglich, es führt aber auch dazu, dass man bei der Anfragebearbeitung in der Regel mehrere „containing $\sim$“, „contained $\sim$“ und/oder „overlapping match“ findet und jeweils das *Beste* bestimmen muss. Dieses Vorgehen wird ohne gute Methoden zur Bewertung und Schnittmengen-Maximierung schnell ineffizient.

Die ersten beiden Varianten sind trotz ihrer genannter Vorteile nur in wenigen Situationen praktikabel. Die Nachteile, die in geringer Effizienz und schlechter Speicherplatzausnutzung resultieren, sind in den meisten Fällen zu gravierend, um eine sinnvolle Anwendung zu ermöglichen. Die letzten zwei Vorgehensweisen weisen nur unter entsprechenden Voraussetzungen klare Vorteile auf. Die Verwendung von ausschließlich Zeichenketten als Attributbelegungen im Mediator jedoch erschwert eine Umsetzung von Strategie 5, ein praktischer Nutzen ist zudem ebenso nicht zu erkennen. Überlappung in den Regionen ist nur unter anderen Zielsetzungen bezüglich der Umsetzung sinnvoll, will man z.B. einen relativen Ausgleich zwischen für die Suche im Cache benötigter Zeit und Netzlast schaffen.

	resultierende Region(en)	**positiv**	**negativ**
1	$q \vee c$ ($\hat{=} Q \cup C$)	intuitiv	große Regionen, evtl. Disjunktionen, schlechte Platzausnutzung
2	$(q \wedge \neg c), (q \wedge c), (c \wedge \neg q)$	feine Granularität	Regionen zu klein, Mehraufwand, Teilung nötig
3	c $(q \wedge \neg c)$	keine Teilung nötig, Ersetzung mit Bezug auf Anfragezeitpunkt möglich	Häufigkeit der Anfrage nicht erfassbar, evtl. Duplikateliminierung
4	q $(c \wedge \neg q)$	Ersetzung mit Bezug auf Anfragehäufigkeit möglich	Teilung nötig
5	$(q \wedge \neg c'), (c \wedge \neg q')$ (Teilung v. $Q \cap C$)	Nutzen bei Zahlenbereichen	fehlender Nutzen für YACOB
6	q, c (Überlappung)	Überlappungen möglich	Mehraufwand in Anfragebearbeitung

Tabelle 2: Varianten der Behandlung semantischer Regionen (siehe Abbildung 10)

Unter den gegebenen Gesichtspunkten verbleiben Verfahren 3 und 4 als wählbare Alternativen. Welche Wahl im Endeffekt getroffen wird, hängt primär von umzusetzender Ersetzungsstrategie ab. Tabelle 2 fasst die auf Seite 36 genannten und daraufhin geschilderten Möglichkeiten mit den Vor- und Nachteilen einer Anwednung im YACOB-Mediatorsystem noch einmal zusammen und nimmt dabei Bezug auf Abbildung 10.

Die Auswahl der Strategie zur Verwaltung der semantischen Regionen hat starken Einfluss auf die Cache-Verwaltung, die Speicherplatzausnutzung und die Effizienz der Anfrageverarbeitung. Im vorliegenden Anwendungsfall üben zwei Punkte maßgeblichen Einfluss auf die Entscheidung bezüglich der umgesetzten Variante aus: die umgesetzte Ersetzungsstrategie und die Tatsache, dass durch die zu Grunde liegenden Wertebereiche auf Zeichenketten und die dadurch hauptsächlich in den Prädikaten enthaltenen Tests auf Zeichenketten-Gleichheit Überlappungen eher selten auftreten werden.

In der vorliegenden Implementierung werden semantische Regionen generell in ihrem Ursprung belassen (d.h. entsprechend der zugehörigen Anfrage). Alle Regionen sind untereinander disjunkt und werden durch eine Formel aus nur konjunktiv verknüpften Konstantenselektionen beschrieben. Überlappungen sind ausgeschlossen, so ist auch eine effiziente Suche nach dem größtmöglichen Teil der im Cache vorliegenden Ergebnisdaten möglich. Die zusätzlich bei den Quellsystemen angefragten Daten bilden daraufhin eine oder mehrere neue semantische Regionen im Cache. Dabei entstehen mehrere Regionen, wenn die Komplementäranfrage nicht mehr ohne Disjunktionen zu formulieren ist. Ein alternatives Vorgehen ist, die Originalanfrage anstatt der konstruierten Komplementäranfrage zur Beantwortung an die Quellen zu leiten. Mögliche Gründe dafür werden in Abschnitt 5.3 genau geschildert, hier sei nur erwähnt, dass die resultierende Anfrage zu komplex werden kann. Dieses Vorgehen verlangt nach Erhalten der Ergebnisdaten eine Duplikateliminierung, um die Daten in den neuen semantischen Regionen nicht redundant zu speichern. Die semantische Beschreibung der gebildeten Regionen muss dann allerdings auch den tatsächlich darin erfassten Daten entsprechen.

Zur Diskussion über die Form und Probleme bei der Bildung wird wiederum auf Abschnitt 5.3 verwiesen. Dort wird die Bildung der komplementären Anfrage beschrieben, die bei beiden geschilderten Vorgehensweisen die Beschreibung der neu entstehenden semantischen Regionen darstellt, auch wenn sie im zweiten Fall nicht in dieser Form an die Quellsysteme weitergeleitet wird. Wichtig ist hier einzig die Beschränkung auf konjunktive Verknüpfungen in den Formeln und die Vermeidung von Duplikaten in verschiedenen Regionen, beides Voraussetzungen für die Verwaltung der Regionen sowie die Anfragebearbeitung.

Umgesetzt wird also Variante 3 der in diesem Abschnitt beschriebenen möglichen Vorgehensweisen. Da semantische Regionen nie geteilt werden und die erfassten Daten somit immer der ursprünglichen Anfrage entsprechen, wird die Realisierung eines Ersetzungsverfahrens basierend auf dem Alter der Cache-Einträge ermöglicht. Umformungen einmal gebildeter Prädikate und Modifikationen entsprechender Ergebnisverweise werden dadurch ebenfalls unnötig, was die Cache-Verwaltung vereinfacht. Da Vor- und Nachteile der einzelnen Möglichkeiten im Bezug zum YACOB-Mediatorsystem die umzusetzende Lösung schon auf Variante 3 und 4 einschränkten, ist es hier also die gewählte Ersetzungsstrategie, die zur endgültigen Entscheidung führt. Ausführungen dazu finden sich in Abschnitt 4.5. Ein weiterer Vorteil des gewählten Verfahrens ist das ermöglichte effiziente Vorgehen bei der Suche nach gespeicherten Ergebnisdaten.

Treten Überschneidungen zwischen den Ergebnismengen bearbeiteter Anfragen und den Ergebnismengen gespeicherter Cache-Einträge auf, so entstehen im Cache, bedingt durch die disjunkte Speicherung der Daten, untereinander zusammenhängende semantische Regionen. Erst die Vereinigung der Daten mehrerer solcher Regionen entspricht der Ergebnismenge der ausgeführten Anfrage. Folgen weitere, Ergebnismengen erweiternde Anfragen, so müssen die Daten dieser Regionen jeweils erneut vereint und die komplementären Daten in den Cache geschrieben werden. Während der Anfragebearbeitung werden die Beziehungen zwischen diesen zusammenhängenden semantischen Regionen allerdings für jede ausgeführte Anfrage erneut durch interne Prozeduraufrufe bestimmt, anstatt die einmal gewonnen Informationen sinnvoll zu speichern und im weiteren Verlauf zu nutzen. Mit steigender Anzahl solch untereinander verknüpfter Regionen steigt somit die auch der Aufwand der Anfragebearbeitung und die für die Suche im Cache benötigte Zeit erhöht sich. Da Überlappungen zwischen Anfragen beim YACOB-Mediatorsystem durch die vorwiegende Verfeinerung der Ergebnisdaten und die Beschränkung auf Zeichenketten bei den Wertebreichen der Attribute eher selten auftreten, kann diese Problematik bei der Entwicklung des semantischen Caches als weniger bedeutend eingestuft werden. Für den Fall vermehrt auftretender Überlappungen wird hier kurz ein Vorschlag zur erweiterten Verwaltung der semantischen Regionen skizziert, der in diesem Fall eine Effizienzsteigerung bezüglich des umgesetzten Verfahrens verspricht.

Das Verfahren basiert auf der hierarchischen Verwaltung der semantischen Regionen in einer Baumstruktur. Überlappende oder ineinander enthaltene Regionen bilden zusammenhängende, virtuelle Regionen auf einer höheren Stufe der Hierarchie. Solch virtuelle Regionen enthalten keine eigentlichen Daten, sondern Verweise zu den „Kind“-Regionen.

Die semantischen Regionen, die schließlich auf Einträge im physischen Speicher verweisen und zur Bildung der Ergebnismenge vereint werden müssen, befinden sich an den Blättern des Baumes. Dieses Vorgehen erfordert, dass existierende Regionen geteilt werden und mehrere Verweise auf ein und die selbe Region möglich sind. Datenredundanz wird durch das Nutzen der logischen Verweise vermieden. Die Beziehungen zwischen den im Cache gespeicherten Datenmengen werden somit explizit erfasst. Es entstehen mehrere, untereinander wiederum disjunkte Bäume, die im Cache zusätzlich zu verwalten sind. Der Effekt ist eine beschleunigte Anfragebearbeitung, denn viele (kleine) Regionen können aus der Betrachtung ausgeschlossen werden. Die Suche im Cache beginnt nun an den Wurzeln der einzelnen Bäume. Dabei kann auf jeder Stufe entschieden werden, ob die enthaltenen Teilbäume einen Teil der Ergebnisdaten umfassen oder nicht. Bildet auf einer Stufe die komplette (virtuelle) Region ein Teil der oder die gesamte Antwort, so kann man die Ergebnisdaten mit entsprechenden Methoden aus den zughörigen Blatt-Regionen extrahieren. Erkennt man andererseits ein „disjoint match“ gegen eine (virtuelle) Region, so kann die Suche für den gerade betrachteten Baum auf dieser Stufe abgebrochen werden. Noch nicht betrachtete Teilbäume und die entsprechenden semantischen Regionen können keine Teile der Ergebnisdaten enthalten. Während der Suche muss die bearbeitete Anfrage also gegen weniger einzelne Regionen geprüft werden. So muss man unter anderem bei Bearbeitung einer Anfrage nie mehrere „contained match“ innerhalb eines Baumes feststellen, denn liegen mehrere enthaltene Cache-Einträge vor, so wird deren Vereinigung durch eine virtuelle Region repräsentiert und auf Stufe dieser Region wird der Zusammenhang bereits erkannt. Die Nutzung solcher Baumstrukturen kann somit die Anzahl zu untersuchender Cache-Einträge drastisch einschränken und dadurch die Anfragebearbeitung merklich beschleunigen. Ein positiver Effekt ist allerdings nur bei recht vielen überlappenden Anfragen absehbar. Der zusätzliche Mehraufwand für die Baumverwaltung könnte sonst leicht die erzielte Beschleunigung überschreiten. Die physische Speicherung der Baumstrukturen kann man gut an das verwendete Datenformat anpassen, indem man den graphbasierten Charakter des XML-Datenmodells ausnutzt. Ähnliche Ansätze zur Erhöhung der Performanz in semantischen Caches durch hierarchische Verwaltung der Einträge finden sich z.B. in Form *Trie*-basierten Cachings ([HS03]) oder inkrementeller Zugriffsmethoden für *ViewCache* ([Rou91]).

4.4 Physischer Speicher

Im YACOB-Mediatorsystem werden alle globalen Anfragen zunächst in CQuery formuliert. Für das Caching relevant ist die XPath-Komponente im **`WHERE`**-Teil der Anfrage. Von den Quellsystemen bzw. ihren Wrappern erhaltene Daten werden konsequent immer in XML-Format repräsentiert und innerhalb des Mediators auch in dieser Form verarbeitet. Im Cache kann man sich somit auf das Datenformat XML und die zugehörige Anfragesprache XPath konzentrieren.

Die Anforderungen an die physische Speicherung von Anfragen und Ergebnis-Dokumenten im Cache, der unabhängig von den Quellen als Teil der Zugriffskomponente des Mediators umgesetzt wird, liegen somit auf der Hand.

XML-(Teil-)Dokumente, die von den Quellen gelieferte Ergebnisdaten enthalten, müssen inklusive ihrer semantischen Beschreibung im Cache gespeichert und bei Bedarf wieder an das System übergeben werden. Dabei soll die Bestimmung und Bildung der im Cache vorliegenden Ergebnisdaten, welche in der Regel aus mehreren Teilen gespeicherter Dokumente zusammengesetzt sind, ohne große Verzögerung in der Anfrageverarbeitung erfolgen. Der Zugriff auf Teile der persistent und ausfallsicher zu speichernden Daten muss demnach in angemessener Zeit möglich sein.

Die Speicherung von XML-Daten und ein effizienter Zugriff darauf werden mit zunehmender Verwendung dieses Formates in unzähligen Anwendungen immer bedeutender. In vielen Fällen, wie auch in diesem, sind dabei auch die typischen Eigenschaften von Datenbanksystemen wünschenswert oder gar notwendig. Die im vorliegenden Fall hauptsächlich geforderten Aspekte sind dabei, wie schon erwähnt, die persistente Speicherung, Ausfallsicherheit und ein effizienter Zugriff auf Teile der Daten (effiziente Anfrageunterstützung). Mit steigendem Bedarf an Datenbanksystemen zur Verwaltung von XML-Daten wächst auch die Zahl von Beiträgen und Lösungen zu diesem inzwischen viel untersuchten Themengebiet. Ansätze reichen von Abbildung auf (objekt-)relationale Daten und Speicherung in bekannten Systemen (eventuell mit XML-Erweiterungen) bis hin zu nativen XML-Datenbanken, die vom Grunde an entwickelt wurden, nur um XML-Daten zu speichern. Wesentlichen Einfluss auf die Wahl der Speicherungsmethode haben dabei die Dokumenteigenschaften (daten- oder dokumentorientiert, Mischfälle) und der Anwendungskontext. Mögliche Speicherungsvarianten teilt man drei prinzipiellen Ansätzen zu: Die Speicherung von Dokumenten als Ganzes, die Dekomposition und generische Speicherung (z.B. auf Basis eines Graphmodells) und die strukturierte Speicherung in Datenbanken (dokumentspezifische Abbildungen auf Datenbankschemata). Desweiteren unterscheidet man Formen der hybriden Speicherung, die Ansätze aller drei Klassen kombinieren. Einen guten Überblick über derzeit existierende Lösungen und allgemeine Aspekte des *XML Data Management* bietet das gleichnamige Buch [CRZ03] sowie [KMRU02].

Für den vorliegenden Fall ist eine Speicherung der XML-Dokumente als Ganzes anzustreben. Die Dokumente müssen im Cache nicht verändert werden, sondern werden ganz oder teilweise angefragt. Die Speicherungsmethode sollte die Erstellung von Indexen auf den Inhalten und der Struktur der Dokumente ermöglichen, um einen effizienten Anfragemechanismus zu gewährleisten. Als Anfragesprache sollte XPath oder XQuery unterstützt werden. Die Verwendung herkömmlicher DBS und entsprechende Abbildung der XML-Daten bringt hier keine Vorteile. Zudem ist dieses Vorgehen in der Regel komplizierter und aufwendiger als die Speicherung von Dokumenten als Ganzes, weshalb diese Möglichkeiten hier nicht weiter in Betracht gezogen werden.

Im semantischen Cache des YACOB-Mediatorsystems wird eine native XML-Datenbank mit effizienter XPath-Anfrageunterstützung zur physischen Speicherung der Daten eingesetzt. Auch auf diesem jungen Gebiet der Datenbanken ist die Auswahl der möglichen Systeme schon recht groß. Die in der Implementierung verwendete Datenbank wird in Kapitel 6 vorgestellt. Dort werden die Vorteile einer nur für XML entwickelten Datenbank im Allgemeinen und die der eingesetzten Datenbank im Speziellen behandelt.

4.5 Cache-Kohärenz und Ersetzungsstrategie

Jeder Cache ist in seinem Platz limitiert. So kann es vorkommen, dass bei Anforderung eines neuen Datenobjektes andere Objekte aus dem Cache entfernt, also verdrängt werden müssen. Wie oft das nötig ist und wie groß die Bedeutung einer optimalen Ersetzungsstrategie ist, wird maßgeblich durch die Cache-Größe beeinflusst. Das Cache-Kohärenzproblem hingegen beschreibt den Umstand, dass Kopien der Daten im Cache veraltet sind, werden sie auf Seite der Quellen verändert. Ein Protokoll zur Kohärenzsteuerung muss solche Situationen erkennen und entsprechende Daten gegebenenfalls invalidieren. Dieser Abschnitt beschreibt kurz bekannte Verfahren zur Verdrängung von Daten und Wahrung der Kohärenz, nennt besondere Aspekte des Web-Caching, die Einfluss auf die umgesetzten Verfahren nehmen und beschreibt schließlich die im Cache des YACOB-Mediatorsystems umgesetzte Lösung. Eine ausführlichere Betrachtung der hier erwähnten Aspekte behandelt [Wei02].

Die Cache-Größe nimmt wesentlichen Einfluss auf das Leistungsverhalten des Caches. Dabei nimmt nicht nur die erreichte Zugriffszeit auf Datenobjekte Einfluss auf die optimlae Größe, sondern auch die mit dem Datenzugriff verbundenen Kosten. So sind die Kosten für den notwendigen Speicherplatz eines Hauptspeicher-Caches wesentlich größer als z.B. bei einem plattenresidenten Proxy-Cache, denn RAM-Preise überschreiten die Preise für Magnetplatten um ein Vielfaches. Zur Berechnung entstehender Kosten-Leistungs-Verhältnisse gibt es einfache Überschlagsverfahren, wie z.B. die 5-Minuten-Regel, die sich am Datendurchsatz als Leistungsmetrik orientiert. Die antwortzeitorientierte Faustregel hingegen orientiert sich an der Antwortzeit als Leistungsmetrik. Gerade letztere suggeriert eine Cache-Größe als optimal, bei der es auf die Ersetzungsstrategie kaum noch ankommt. In der Praxis allerdings gibt es so viele Unwägbarkeiten, dass die Anwendung solcher Faustregeln nicht immer praktikabel ist. Oft arbeitet man dann mit Wunschtrefferraten oder Schranken für die Antwortzeit, was häufig die Anwendung anspruchsvoller mathematischer Modelle nötig macht.

Bekannte Ersetzungsverfahren, die auch in Datenbanksystemen, Dateisystemen und anderen Caching-Ansätzen genutzt werden, sind *LRU (least recently used)*, *LFU (least frequently used)* und Verallgemeinerungen dieser Verfahren. Alle Ersetzungsstrategien weisen den Datenobjekten Werte zu, die das Zugriffsverhalten der Vergangenheit widerspiegeln und zur Schätzung des Zugriffsverhaltens in der Zukunft genutzt werden. Diese *Wichtigkeit* (auch: Popularität, Priorität, Hitze) wird zur Realisierung der Ersetzungsstrategie genutzt. Wird eine Verdrängung von Datenobjekten nötig, so werden die Objekte mit der geringsten Wichtigkeit zuerst aus dem Cache entfernt. Bei Verfahren der LRU-Familie nimmt dabei das Alter des Objektes, bei Verfahren der LFU-Familie die Häufigkeit des Zugriffs Einfluss auf diese Werte. Dabei wird unter dem Alter eines Objektes nicht das Datum des Einlagerns im Cache verstanden, sondern die seit dem letzten Zugriff vergangene Zeit. Eine Strategie die nur auf den Zeitpunkt des Ladens des Objektes Bezug nimmt, würde auf eine *FIFO*-Strategie führen (*First In First Out*) und ist in den meisten Anwendungsfällen für Caching unbrauchbar. Beide Ansätze weisen aber auch Probleme auf: So passen sich LFU-Strategien nur sehr langsam zeitlichen Veränderungen an, d.h. Objekte, die in weiter Vergangenheit eine hohe Zugriffshäufigkeit aufwiesen, jetzt aber nicht mehr populär sind, bleiben unnötig lange im Cache.

LFU-Strategien hingegen weisen Objekten, auf die nur einmal, aber gerade eben zugegriffen wurde, fälschlich eine hohe Wichtigkeit zu. Lösung bietet eine Modifikation der Verfahren oder aber die Kombination der beiden Kriterien. Als weiteres Verfahren dieser Familie ist z.B. LRU-k zu nennen, bei dem die Entscheidung anhand des k-ten letzten Zugriffs auf das Objekt getroffen wird.

Im Zusammenhang mit Web-Caching gewinnt neben der zeitlichen Lokalität, auf der die geschilderten LRU- und LFU-Verfahren basieren, auch die örtliche Lokalität gespeicherter Datenobjekte an Bedeutung. Daten auf weit entfernten oder schlecht erreichbaren Servern ist eine höhere Priorität zuzuweisen als z.B. Daten aus einem lokalen Netzwerk. Auch andere Faktoren, wie die Größe der gespeicherten Objekte oder die Kosten des Zugriffs auf die Quellsysteme, nehmen speziellen Einfluss auf die Ersetzungsstrategie. Bekannte Lösungen sind z.B. temperaturorientierte Verfahren, die die Wichtigkeit eines Datenobjektes auf die einzelnen Bytes des Objektes umrechnen, also die Objektwichtigkeit entsprechend der Objektgröße normalisieren. Die Einbeziehung der Zugriffskosten wird oft als mathematisches Optimierungsproblem formuliert. Natürlich stellt es einen erheblichen Aufwand dar immer, wenn eine Verdrängung angezeigt ist, ein solches System zu lösen. Vielmehr müssen Temperatur und Kosten dynamisch geschätzt werden, z.B. mit Hilfe von Online-Statistiken. Sogenannte nutzenorientierte Caching-Verfahren verbinden zur Berechnung der Wichtigkeit eines Objektes die Temperatur- und Kostenfaktoren mit den bei Behandlung der Basisstrategien vorgestellten Prioritäten.

Verfahren zur Lösung von Kohärenzproblemen unterscheidet man nach zwei Ansätzen: *Push-basierte* und *Pull-basierte* Verfahren. Bei Push-basierten Strategien übernimmt die Server-Seite die aktive Kontrolle der Kohärenz, protokolliert also von Clients gelesene Daten und versendet Nachrichten zur Aktualisierung bzw. Invalidierung dieser Daten. Eine solch aktive Rolle der Quellsysteme wird im YACOB-System allerdings nicht unterstützt, weshalb man sich hier auf die Pull-basierten Ansätze beschränken kann. Bei diesen Strategien bemüht man sich nur auf Cache-Seite um eine aktive Aktualisierung der Datenobjekte. Mögliche Lösungen werden beim Web-Caching in der Regel weiter vereinfacht, da eine Änderung der Daten in den meisten Fällen nur auf Seite der Quellen erfolgen kann. Systeme, in denen Änderungen auf jeder beliebigen Kopie initiiert werden können, machen wesentlich komplexere Verfahren nötig, als z.B. der vorliegende Anwendungsfall. Die in HTTP (*HyperText Transfer Protocol*) vorgesehene Standardlösung ist die Nutzung der Methode *Get-If-Modified-Since*, mit der Daten bei den Quellsystemen konditional angefordert werden können. Hat sich das Objekt seit dem beim Aufruf mitgelieferten Parameter verändert, wird eine aktuelle Kopie von der Quelle versendet. Um die Netzlast zusätzlich zu reduzieren, wird diese Methode meist in Zusammenhang mit einem Schätzwert *TTL (Time To Live)* verwendet. Dieser Wert approximiert die Zeit bis zur nächsten Änderung des Datenobjektes, vor Ablauf dieser Zeit wird nicht auf eventuelle Änderungen geprüft. Die Struktur des YACOB-Systems und der integrierten Quellen schließt die Nutzung einer Methode wie *Get-If-Modified-Since* jedoch aus. Auf Änderungen an den Daten kann nur geprüft werden, indem die Daten neu angefordert werden. Die Alternative ist hier also die Entwicklung einer Pull-Strategie, die einen von den zu erwartenden typischen Quelleneigenschaften abhängigen TTL-Wert nutzt und auf Änderungen nur durch neue Anfragen prüfen kann.

Die Anforderungen an Ersetzungsstrategien bei der Datenintegration im Web unterscheiden sich etwas von denen anderer Web-Anwendungen ([CRS99]). Zudem ermöglicht semantisches Caching die Unterstützung angepasster Ersetzungsstrategien. So ist es möglich, zu ersetzende Objekte anhand spezieller Lokalitätsaspekte auswählen, wie z.B. ihrer semantischen Distanz untereinander ([DFJ+96]). Solche Kriterien passen sich dynamisch an die ausgeführten Anfragen an und beruhen nicht auf der statischen Zusammengehörigkeit von Daten, wie es z.B. beim Seiten-Caching der Fall ist. Die semantischen Regionen, in denen die gespeicherten Daten gruppiert sind, können aufgetrennt werden und die populäreren Teile können in einer neuen Region mit höherer Wichtigkeit vereint werden. Im Zusammenhang mit Caching in mobilen Anwendungen wird z.B. die vom Nutzer zu erwartende Bewegungsrichtung mit geographischen Parametern beim Eintragen der Daten in den Cache verbunden ([RD00, Ren00]). Andere Ansätze sind z.B. dynamisches LRU, kurz D-LRU ([RD98]), oder aber die explizite Unterscheidung in populäre und unpopuläre Anfragen und dementsprechende Behandlung ([LHK01]). Die Kohärenz von Daten wird vor dem Anwendungshintergrund der Informationsbereitstellung oft nur anhand des Alters geprüft. Der Charakter der integrierten Bibliotheken und anderen reinen Informationsquellen macht die Betrachtung von möglichen Änderungen an bestehenden Daten in vielen Fällen überflüssig.

Im Fall des YACOB-Mediators kann man von relativ geringen Speicherplatzkosten für den Cache ausgehen, da es sich beim physischen Speicher um ein magnetplattenbasiertes XML-DBS handelt. Die Cache-Größe wird hier nur durch Systemparameter begrenzt. Eine geringe Antwortzeit ist durch die Nähe des Caches zum Mediator auch bei einem relativ großen Cache gegeben. Die gewählte Ersetzungsstrategie (trotz seiner Größe ist der Cache keine vollständige Kopie der Quellsysteme, Objekte müssen gegebenenfalls also auch hier verdrängt werden) verliert damit an Bedeutung. Die Bedeutung einer effizienten Unterstützung der interaktiven Anfrageverfeinerung im Mediator rückt die Lösung das Cache-Kohärenzproblems zusätzlich in den Vordergrund. Man kann davon ausgehen, dass Daten generell zur Wahrung der Kohärenz aus dem Cache entfernt oder aktualisiert werden. Eine nötige Verdrängung von Daten aufgrund von Speicherplatzproblemen im Cache ist eher selten zu erwarten.

Die im Cache gespeicherten Daten sind in gegenseitig disjunkten semantischen Regionen gruppiert. Dies ermöglicht ein Vorgehen auf Basis dieser Regionen anstatt einzelner Daten. Informationen und Werte bezüglich der Wichtigkeit werden für jede Region einzeln erfasst. Sind die Kriterien erreicht, die eine Verdrängung aus dem Cache anzeigen, so werden alle der entsprechenden Region zugeordneten Daten aus dem Cache entfernt. Wie in Abschnitt 4.3 erläutert wurde, führen zu größe Regionen bei der Ersetzung zu schlechter Speicherplatzausnutzung Wird eine Region aus dem physischen Speicher entfernt, so entfernt man einen Großteil aller gespeicherten Daten. Andererseits führt eine hohe Anzahl verhältnismäßig kleiner Regionen zu einer aufwändigeren Anfragebearbeitung und somit zu Laufzeiteinbußen. Es wird zwar ein Ersetzungsverfahren auf Basis einer viel feineren Granularität ermöglicht, jedoch müssen im Verlauf der Anfragebearbeitung zu viele Regionen berücksichtigt werden.

Auf Kohärenz der Daten wird im entwickelten Cache nur anhand von TTL-Werten geprüft. Die semantischen Regionen im Cache werden mit zwei Zeitstempeln versehen.

Ein Zeitstempel gibt den Zeitpunkt der Kollektion an (T_K), ein zweiter Zeitwert wird bei jedem Zugriff aktualisiert (T_L). Überschreitet T_L ein Vielfaches der durchschnittlichen Sitzungszeit oder ist der Eintrag entsprechend dem Wert T_K „zu alt“ (ein optimales Alter ist in entsprechenden Experimenten zu ermittteln), so wird er zu gegebenem Zeitpunkt aus dem physischen Speicher entfernt. Das maximale Alter T_K eines Cache-Eintrages ist vom Charakter des WWW und der integrierten Quellsysteme abhängig zu machen. Da Änderungsoperationen im Mediatorsystem nicht berücksichtigt werden müssen, werden Daten im Cache nur inkohärent bei Änderung der Daten in integrierten Quellen oder bei Integration einer neuen bzw. Ausschluss einer bisherigen Quelle. Änderungen in Daten bisheriger Quellsysteme sind durch den Charakter der eingebundenen Kulturdatenbanken nicht sehr häufig und durch gute Heuristiken bezüglich des Alters der Cache-Einträge leicht erfassbar. Das zu lösende Hauptproblem ist also, dass die Daten im Cache eventuell unvollständig aufgrund von Objekten sind, die auf Seite der Quellsysteme hinzugefügt wurden. Eine Änderung der Anzahl unterstützter Quellen könnte dem Cache signalisiert und entsprechende Einträge aktualisiert oder entfernt werden. Zur Wahrung der Kohärenz ist T_K somit der Faktor mit dem wesentlichen Einfluss, T_L unterstützt eher eine Verdrängungsstrategie. Das Entfernen von Datenobjekten, die über längere Zeit nicht gelesen wurden, unterstützt aber in gewisser Weise auch die Vermeidung von Kohärenzproblemen.

Der Bezug auf den Zeitpunkt des Lesens eines Datenobjektes und auf den Zeitpunkt des letzten Zugriffs zur Lösung von Kohärenzproblemen deutet schon an, dass die Ersetzungsstrategie hier den Anforderungen entsprechend zunächst recht einfach umgesetzt wurde. Das Verfahren verknüpft den FIFO-Ansatz mit der LRU-Strategie. Sollten Daten einmal aufgrund neu in den Cache aufzunehmender Objekte entfernt werden müssen, so werden zunächst die laut T_L am längsten ungenutzten Daten entfernt. Ist der Wert von T_L für mehrere semantische Regionen gleich, so erfolgt die Wahl im Bezug auf das von T_K angegebene Alter der enthaltenen Daten.

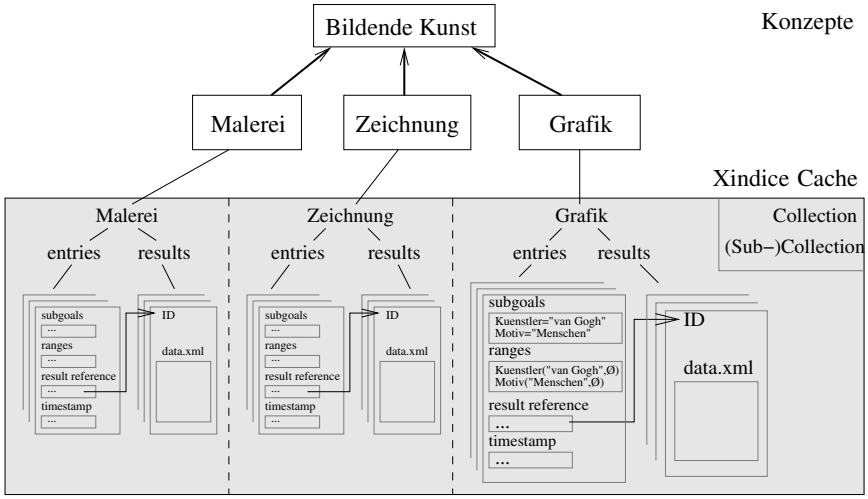
Trotz dieses recht einfachen Vorgehens ist das umgesetzte Verfahren absolut ausreichend, denn Hauptanforderung an den Cache ist die effiziente Unterstützung interaktiver Sitzungen. Die Aktualität der Daten im Cache bei Anfragen, die Ergebnismengen unmittelbar nach Erhalt einschränken oder erweitern, wird durch Nutzung der Zeitstempel ausreichend gewährleistet.

4.6 Anbindung an das Konzeptmodell

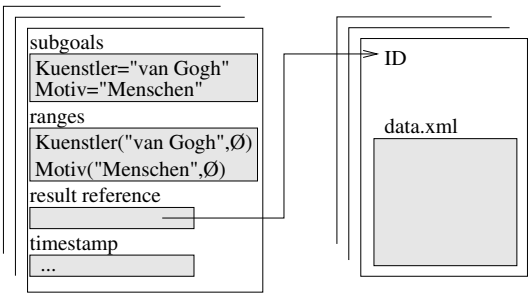
Um die Struktur des Caches und seine Anbindung an das Konzeptmodell abschließend noch einmal zu verdeutlichen, zeigt Abbildung 11 auf Seite 46 einen Teil des Konzeptschemas mit einem zugehörigen Eintrag. Dieser Cache-Eintrag wird erzeugt, wenn die XPath-Anfrage

```
//Grafik[Kuenstler='van Gogh' and Motiv='Menschen']
```

ausgeführt und das zugehörige Ergebnis als Ergebnis-Dokument data.xml im Cache gespeichert wird.



(a) Struktur des Caches



(b) Cache-Eintrag

Abbildung 11: Struktur des semantischen Caches

Die abgebildeten Verwaltungseinträge sind nur beispielhaft und nicht vollständig. Erläuterungen zur internen Struktur des Caches, die hier mit abgebildet ist, folgen in Abschnitt 6.1. Dort werden die für die Implementierung wichtigen Design-Entscheidungen erläutert. Eine Beschreibung der zur Verwaltung der Cache-Einträge nötigen Informationen erfolgt bei Vorstellung der zur Anfragebearbeitung verwendeten Algorithmen in Kapitel 5. Hier liegt das Augenmerk auf der dargestellten Anbindung an die Ontologie.

5 Anfragebearbeitung

In diesem Kapitel wird, in Anbindung an die Erläuterung der Cache-Struktur im letzten Kapitel und an die in Abschnitt 2.2 beschriebene Anfragebearbeitung im Mediatorsystem, der Ablauf der Anfragebearbeitung im Cache erläutert. Zunächst wird der in Abschnitt 3.4 vorgestellte Algorithmus zur Lösung des *Query Containment* an die im Mediatorsystem unterstützten Wertebereiche auf Zeichenketten (in den folgenden Abschnitten wird auch der Begriff *Wort* für eine Zeichenkette verwendet) angepasst. Es folgt die Beschreibung des Vorgehens zur Suche nach passenden Einträgen im Cache sowie Erläuterungen zum Bilden der Komplementäranfrage. Spezielle Anforderungen, die durch Unterstützung von CQuerys Textähnlichkeitsoperator entstehen, werden am Ende des Kapitels behandelt.

5.1 Query Containment auf Wertebereichen von Zeichenketten

In Abschnitt 3.4 wurde ein Algorithmus zur Lösung des *Query Containment* in einem allgemeinen Fall auf ganzzahligen Wertebereichen vorgestellt. Parallel wurde angemerkt, dass der vorliegende Anwendungsfall einige Veränderungen nötig macht. Prinzipiell gelten auch hier die in Abschnitt 3.3 erfassten Definitionen der Beziehungen zweier Anfragen und allgemeinen Einschränkungen. Die konkreten Unterschiede in den Annahmen werden in der folgenden Auflistung zusammengefasst.

Die Annahmen, die in beiden Fällen gelten, sind:

i) Das *Query Containment* wird beschränkt auf Prädikate, deren Teilbedingungen alle konjunktiv verknüpft sind.

ii) Es treten nur Bedingungen der Form '$A\ \theta\ c$', A ist Attribut, c Konstante und $\theta \in \Theta$, auf.

iii) Die Anzahl der Attribute ist von vornherein bekannt.

Die speziellen Annahmen im vorliegenden Anwendungsfall sind folgende:

i) Statt auf ganzen Zahlen wird nun auf Wertebereichen von Zeichenketten gearbeitet, alle Attribut-Belegungen werden als Zeichenketten verstanden.

ii) Die Menge zugelassener Operatoren Θ wird an die veränderten Wertebereiche angepasst: der zuvor festgelegten Menge zulässiger Operatoren Θ wird CQuerys spezieller Textähnlichkeitsoperator '$\sim=$' und der Ungleichheitsoperator '$\neq$' hinzugefügt, es gilt somit $\Theta = \{<, \leq, =, \geq, >, \neq, \sim=\}$. Im Weiteren Verlauf werden es vor allem die neu hinzugefügten Operatoren '$\neq$' und '$\sim=$' sein, deren Behandlung den Aufwand der Anfragebearbeitung maßgeblich beeinflusst.

Die Einschränkung auf Zeichenketten rührt einzig aus der geplanten Anwendung im YACOB-Mediatorsystem.

Im Mediator wird nur die Suche auf Zeichenketten unterstützt. Die zu Grunde liegenden Prinzipien sind, wie Abschnitt 3.4 gezeigt hat, ohne weiteres auch auf ganzzahlige Wertebereiche anwendbar. Die Form der Suche und die Charakteristika der Ergebnisdaten macht Wertebereiche ganzer Zahlen in der Anwendung allerdings unsinnig. So weisen die Objekte in den Quellsystemen z.B. zum Attribut 'Datum', bei dem Zahlenwerte als Attributbelegungen sinnvoll erscheinen, Ausprägungen der Form 'um 1600', '19. Jh.' usw. auf. Selbst diesem Attribut sind hier also Zeichenketten zuzuordnen.

In Abbildung 12 ist der Ablauf des Algorithmus ohne die Anweisungen der inneren **for**-Schleife abgebildet. Diese sind separat in Abbildung 14 auf Seite 53 dargestellt und werden dort behandelt.

Gegeben:
Query q
CachedQuery c
Rückgabe:
match-type zwischen q und c

```
1       exact = true;
2       containing = false;
3       contained = false;
4       rq = build-ranges(q);
5       if rq = NULL then return "q unsatisfiable";
6       rc = build-ranges(c);
7       if rc = NULL then return "c unsatisfiable";
8       for i = 1 to #attr do
...         innere for-Schleife zur Überprüfung der ranges
9       od
10      if exact = true then return "exact";
11      if containing = true && contained = false then return "containing";
12      if containing = false && contained = true then return "contained";
13      return "overlapping";
```

Abbildung 12: Prozedur *match-type* für Zeichenketten, ohne innere **for**-Schleife

Die Abbildung verdeutlicht die Gemeinsamkeiten mit dem Algorithmus für ganze Zahlen in Abbildung 8 auf Seite 27 und zugleich einen entscheidenden Unterschied: Die bei Beschreibung von Abbildung 8 schon angesprochene mögliche Laufzeitverbesserung wurde umgesetzt. Um die Fälle „containing" und „contained" zu prüfen, ruft man nicht wie bisher den *match-type*-Algorithmus jeweils erneut mit einer der Anfragen in negierter Form auf, sondern integriert diese Überprüfung mit der Kontrolle auf „disjoint" und „exact" in nur einen Lauf über die *ranges* aller Attribute. Der Unterschied ist in den hier zusätzlichen Zeilen 2 und 3 zu erkennen. Boolesche Variablen[2] werden initialisiert, deren Belegung den jeweils festgestellten Fall signalisiert. Die rekursiven Aufrufe von *match-type* (vorher Zeilen 11 und 12) am Ende der Prozedur zur Bestimmung der Beziehungen zwischen den Anfragen können nun eingespart werden.

[2] Variablen, die nur zwei Werte annehmen können: entweder WAHR ('W') oder FALSCH ('F') bzw. TRUE ('T') oder FALSE ('F')

Stattdessen werden hier die beiden Variablen entsprechend ausgewertet. Ist die Belegung beider Variablen 'W', so liegt der überlappende Fall vor (z.B. „containing“ in einem Attribut, „contained“ in einem zweiten). Wie sich diese Belegung entsprechend der untersuchten Anfragen ändert wird in Abbildung 14 auf Seite 53 und der zugehörigen Erläuterung der inneren **for**-Schleife ersichtlich. Diese und weitere wesentliche Unterschiede zum *Query Containment* auf ganzzahligen Wertebereichen werden in den folgenden Teilabschnitten beschrieben. In der beschriebenen Lösung wird der Operator '$\sim=$' zunächst *nicht* wie oben angegeben erfasst. Die zur Unterstützung von Zeichenketten nötigen Veränderungen werden zunächst nur anhand der Operatormenge $\Theta = \{<, \leq, =, \geq, >, \neq\}$ verdeutlicht. Zur Erfassung des Textähnlichkeitsoperators ist das im Folgenden beschriebene Verfahren mit weiteren Anpassungen anzuwenden. Diese weiteren Veränderungen, die ihnen zu Grunde liegenden Umstände und resultierende Probleme werden, aufgrund ihrer Komplexität, separat in Abschnitt 5.4 behandelt.

Die Prozedur *build-ranges*

Auch im hier vorgestellten Algorithmus sind wieder sogenannte *ranges* zu finden, welche, wie im ersten vorgestellten Algorithmus, die von der Anfrage gestellten Bedingungen an die Attribut-Belegungen repräsentieren. Die zum Erstellen der *ranges* aufgerufene Prozedur *build-ranges* hat hier einen prinzipiell gleichen Ablauf wie im Fall ganzer Zahlen mit Anpassungen an die veränderten Wertebereiche. Ihr Ablauf ist in Abbildung 13 auf Seite 50 abgebildet.

Alle Teilbedingungen '$X_i\ \theta\ c$' werden betrachtet und die *ranges* entsprechend angepasst. Die auffälligste Veränderung ist die integrierte Erfassung des Ungleichheitsoperators in den Zeilen 3, 14 und 17 bis 23. Zusätzlich zu den gebildeten Intervallen enthält die *range* eines jeden Attributes eine Menge von Worten $UnEq$ (*UnEquivalences*), in der in Verbindung mit dem '$\neq$'-Operator angegebene Werte festgehalten werden. Hier repräsentieren, angepasst an die geplante objektorientierte Implementierung des Caches, die *ranges* somit einen strukturierten Datentyp und nicht mehr reine Intervalle. Der Name wurde beibehalten, um den Zusammenhang beider Lösungen zu verdeutlichen.

Im übrigen Teil ist auf den ersten Blick vielleicht kein Unterschied zum Algorithmus auf Seite 28 zu erkennen. Die Abweichungen sind in minimalen Veränderungen in den angewandten Operationen (Zeilen 7, 8, 12, 13 und 16) ersichtlich. Durch die unterstützte lexikographische Ordnung auf Zeichenketten ist es nicht möglich, zu einer Konstante c den Vorgänger *und* den Nachfolger zu bestimmen, einzig der Nachfolger ist, abhängig vom verwendeten Alphabet, ermittelbar. Ein kurzes Beispiel soll das verdeutlichen:

Angenommen, das verwendete Alphabet A umfasst nur die 26 Kleinbuchstaben des lateinischen Alphabets, keine Zahlen, Sonderzeichen oder Großbuchstaben. Zur Zeichenkette $s :=$`gogh` ist der Nachfolger $N_s :=$ `gogha`. Es gibt keine Zeichenkette t, für die $s < t < N_s$ gilt, denn:

- ein solches Wort t muss mit `gogh` beginnen, da sonst gilt $t < s$ (z.B. $t :=$`gogg`) oder $t > N_s$ (z.B. $t :=$`gogi`)

Gegeben:
Query q
Rückgabe:
ranges $(r, UnEq)$, NULL wenn q unsatisfiable

```
1   for i = 1 to #attr do
2       r_i = [-∞, +∞);
3       UnEq_i = ∅;
4   od
5   forall condition 'X_i θ c' of q do
6       switch (θ) do
            /* sei r_i = [a_i, b_i) */
7           case '=': if c < a_i or c ≥ b_i then return NULL;
8                     r_i = [c, c + 1);
9                     break;
10          case '>': r_i = [max(a_i, c + 1), b_i); break;
11          case '≥': r_i = [max(a_i, c), b_i);     break;
12          case '<': r_i = [a_i, min(b_i, c));     break;
13          case '≤': r_i = [a_i, min(b_i, c + 1)); break;
14          case '≠': UnEq_i = UnEq_i ∪ {c};        break;
15      od
16      if a_i ≥ b_i then return NULL;          /* neue Werte!! */
17      if min + 1 = max do                     /* X_i = min in Anfrage enthalten */
18          if min ∈ UnEq_i then return NULL;
19          UnEq_i = ∅;
20      od
21      forall s ∈ UnEq do
22          if s < a_i or s ≥ b_i then UnEq_i = UnEq_i \ {s};
23      od
24  od
25  return (r, UnEq);
```

Abbildung 13: Prozedur *build-ranges* für Zeichenketten

- für jedes Wort t das ebenfalls mit gogha beginnt und mehr Zeichen als N_s enthält ($|t| > |N_s|$) gilt: $t > N_s$

- ebenso gilt $t > N_s$ für jedes dieser t mit der gleichen Anzahl Zeichen wie N_s ($|t| = |N_s|$) und $t \neq N_s$, denn a ist das erste („kleinste“) Zeichen in A.

Der Beweis wird hier offen gelassen. Die Schilderungen sollten genügen, um den Sachverhalt zu verdeutlichen. Der Zusammenhang zwischen der Ordnung auf Zeichenketten und ihrer Länge ist genau das Problem bei der Ermittlung eines Vorgängers. Zu jedem Wort V_s, für das $V_s < s$ gilt, gibt es ein entsprechendes Wort t, so dass $V_s < t < s$ gilt. Erneut wird kein Beweis geführt, zur Konstruktion des entsprechenden Wortes t nimmt man einfach das gewählte V_s und hängt ein beliebiges Zeichen des Alphabetes an.

Da ein solches Element zwischen s und V_s auf diese Weise für *jedes* mögliche V_s gebildet werden kann, ist die Bestimmung eines Vorgängers zu s überhaupt nicht möglich.

Die Folge ist, dass zu beiden Seiten geschlossene Intervalle nicht erstellt werden können: Treten Bedingungen der Form $X_i < c$ auf, so ist zur Erstellung eines geschlossenen Intervalls der Vorgänger von c zu bestimmen, was allerdings nicht möglich ist. Ebenso können nicht zu beiden Seiten offene Intervalle verwendet werden, denn analog wäre bei Bedingungen der Form $X_i \geq c$ wiederum der Vorgänger von c zu ermitteln. Die Lösung ist die Verwendung von Intervallen, die zur „linken“ Seite (min) geschlossen und zur „rechten“ Seite (max) offen sind, denn dadurch ist nur die Bestimmung des jeweiligen Nachfolgers erforderlich. Die Überprüfung der Intervalle (Zeilen 7 und 16) ist dementsprechend angepasst. Die Bestimmung des Nachfolgers eines Wortes c in den Zeilen 8, 10 und 13 wird hier vereinfachend durch $c + 1$ symbolisiert. $-\infty$ und $+\infty$ stellen besondere Werte dar, die den initialisierten Zustand darstellen, ihre Realisierung ist letztendlich eine Frage der Programmierung.

Ist in einer untersuchten Anfrage eine Bedingung b der Form $X_i = c$ enthalten, so überdeckt diese alle anderen Bedingungen bezüglich X_i. Alle weiteren Einschränkungen bezüglich X_i werden, vorausgesetzt sie verletzten in Verbindung mit b nicht die Erfüllbarkeit der Anfrage, nicht in Form von Einträgen in den *ranges* erfasst (Anpassung des Intervalls an c in Zeile 8 und Leeren der Menge $UnEq$ in Zeile 19). Gilt für das zulässige Intervall $[min, max)$ eines Attributes X_i also $min + 1 = max$ (max ist der lexikographische Nachfolger von min), so ist die Bedingung $X_i = min$ oder eine äquivalente Kombination anderer Teilbedingungen, z.B. '$X_i \geq min \wedge X_i \leq min$', in der Anfrage enthalten. In diesem fall gilt für X_i: $UnEq = \emptyset$. Ist umgekehrt $UnEq \neq \emptyset$, so muss $min + 1 \neq max$ gelten. Zusätzlich wird $UnEq$ in jedem Schritt auf nur die Elemente reduziert, die noch im zugelassenen Intervall liegen (Zeilen 21 bis 23). Andere Einträge sind überflüssig, da sie durch das einschränkende Intervall ohnehin nicht erlaubt sind.

Innerhalb der Prozedur wird ebenfalls die Erfüllbarkeits-Eigenschaft der betrachteten Anfrage untersucht. Eine Anfrage ist nicht erfüllbar, wenn für das zulässige Intervall $r_i = [min, max)$ und die Menge $UnEq_i$ eines Attributes X_i gilt:

- $min \geq max$ (Zeile 16) - wie auf ganzzahligen Wertebereichen *nach* der Modifikation der Intervalle überprüft, da in den Zeilen 10 bis 13 immer nur eine Intervallseite verändert wird

- $min + 1 = max$, d.h. die Anfrage enthält $X_i = min$ oder eine äquivalente Kombination anderer Bedingungen, und $min \in UnEq_i$ (Zeile 18) - dieser Punkt kann erst *nach* Anpassung der Intervalle untersucht werden, da erst danach die Einschränkung auf nur einen zulässigen Wert *und* dessen Existenz in $UnEq_i$ erkennbar ist

- $X_i = c$ in Anfrage und c liegt außerhalb des bereits eingeschränkten Intervalls (Zeile 7) - diese Überprüfung muss wiederum *vor* Anpassung der Intervalle erfolgen, da hier wieder beide Seiten des Intervalls verändert und bisherige Einschränkungen somit „überschrieben“ werden

Das Prinzip der hier vorgestellten *build-ranges*-Prozedur entspricht dem der in Abschnitt 3.4 beschriebenen. Die Unterschiede sind die notwendige Verwendung von nur einseitig geschlossenen Intervallen und die Unterstützung des Ungleichheitsoperators. Eine alternative Erfassung von Bedingungen der Form $X_i \neq c$ stellt die Umwandlung in $X_i < c \vee X_i > c$ und die Behandlung der resultierenden Anfrage dar. Da die vorgestellte Lösung des *Query Containment* und die gesamte Cache-Verwaltung aber auf konjunktiven Ausdrücken basiert, würde dieses Vorgehen zur Generierung mehrerer zu behandelnder Teilanfragen führen (entsprechend den einzelnen durch logisches „Oder" ['$\vee$'] verbundenen Teilen der disjunktiven Normalform der resultierenden Anfrage). Jede dieser Teilanfragen wird im Cache separat behandelt, für jede erfolgt somit auch die Bestimmung des *match-type* separat. Von dieser Variante wird aufgrund des zu erwartenden Mehraufwands in der Anfragebearbeitung abgesehen.

Die innere for-Schleife

Abbildung 14 auf Seite 53 schildert den Ablauf der zur eigentlichen Überprüfung der Anfragen realisierten inneren **for**-Schleife.

Am Anfang der Schleife wird für die *ranges* eines jeden Attributes wie bisher geprüft, ob die Intervalle sich gegenseitig ausschließen (Zeile 2). Ist dies nicht der Fall, kann eine genauere Untersuchung der Intervalle inklusive der $UnEq$-Mengen erfolgen. Wird ein Attribut in einer der Anfragen mit '$=$' auf einen Wert festgelegt, so wird dies als Spezialfall behandelt. Zur Erinnerung: dann gilt für das entsprechende Intervall $min{+}1 = max$ (Zeilen 3, 4 und 14). Ist dies in beiden Anfragen der Fall, so werden beide Werte verglichen (Zeile 5) und entsprechend reagiert („disjoint" zurückgegeben oder *exact* bleibt 'W'). Dabei ist dieser Test eigentlich überflüssig, da er schon in Zeile 2 integriert ist. Er dient hier zum leichteren Verständnis des Algorithmus und zur Verbesserung der Lesbarkeit. Ist das Attribut nur in einer der beiden Anfragen auf einen Wert c festgelegt, so schließen sich die Anfragen gegenseitig aus, wenn c in der Menge $UnEq$ der anderen Anfrage existiert (Zeilen 8 und 15). Existiert er dort nicht, so ist die exakt festlegende Anfrage die speziellere (Zeilen 9 und 16). Die Überprüfung, ob c überhaupt im zulässigen Intervall der allgemeineren Anfrage enthalten ist, wurde schon in Zeile 2 integriert.

Sollte keine der Anfragen einen Wert mit '$=$' festgelegt haben, so sind beide identisch, wenn die entsprechenden Intervalle und $UnEq$-Mengen identisch sind (Zeile 20). Sonst ist widerum eine Anfrage allgemeiner, wenn ihr Intervall das Intervall der zweiten Anfrage enthält und ihre $UnEq$-Menge eine Teilmenge der anderen $UnEq$-Menge ist (Zeilen 22 und 25). Tritt keiner der beschriebenen Fälle auf, so überlappen sich die Anfragen (Zeile 28).

Es wird das allgemeine Vorgehen ersichtlich, das anzuwenden ist, wird der $\neq$-Operator in Konstantenselektionen mit in Betracht gezogen. Wiederum gilt, dass das Verfahren erheblich komplizierter wird, werden Verknüpfungen zwischen Attributen ('$A\ \theta\ B$') zugelassen.

Wie bereits geschildert wurde, erfolgt die Überprüfung auf „containing match" und „contained match" am Ende der Prozedur *match-type* durch Kontrolle der Belegung der Booleschen Variablen *containing* und *contained.*

```
1     for i = 1 to #attr do
          /* sei rx_i = ([x_i^min, x_i^max), x_i^UnEq) */
2         if q_i^min ≥ c_i^max or c_i^min ≥ q_i^max then return "disjoint";
3         if q_i^min + 1 = q_i^max then do          /* Attr. i in q mit '=' festgelegt */
4             if c_i^min + 1 = c_i^max then do      /* Attr. i in c mit '=' festgelegt */
5                 if q_i^min ≠ c_i^min then return "disjoint"; /* auch schon in Zeile 2 geprüft */
6             od
7             else do                               /* q in Attr. i spezieller */
8                 if q_i^min ∈ c_i^UnEq then return "disjoint";
9                 containing = true;
10                exact = false;
11            od
12        od
13        else do                                   /* Attr. i in q nicht mit '=' festgelegt */
14            if c_i^min + 1 = c_i^max then do      /* Attr. i in c mit '=' festgelegt */
15                if c_i^min ∈ q_i^UnEq then return "disjoint";
16                contained = true;                 /* c in Attr. i spezieller */
17                exact = false;
18            od                                    /* Attr. i weder in q noch in c */
19            else do                               /* mit '=' festgelegt */
20                if q_i^min = c_i^min and q_i^max = c_i^max and q_i^UnEq = c_i^UnEq then continue;
21                exact = false;
22                if q_i^min ≥ c_i^min and q_i^max ≤ c_i^max and q_i^UnEq ⊇ c_i^UnEq then do
23                    containing = true; continue;
24                od
25                if q_i^min ≤ c_i^min and q_i^max ≥ c_i^max and q_i^UnEq ⊆ c_i^UnEq then do
26                    contained = true; continue;
27                od
28                containing = contained = true;    /* das bedeutet "overlapping" */
29            od
30        od
31    od
```

Abbildung 14: innere **for**-Schleife der Prozedur *match-type* in Abbildung 12

Die Zuordnung der entsprechenden Werte zu diesen Variablen erfolgt in der **for**-Schleife in jedem Schritt bei Kontrolle der jeweiligen Bedingung. Es gilt also, dass eine Anfrage in einer anderen enthalten ist, wenn diese Beziehung für *jedes* bekannte Attribut gilt. Sollte eine Anfrage bezüglich eines Attributes in einer zweiten enthalten sein und sie diese aber gleichzeitig bezüglich eines weiteren Attributes wiederum selbst enthält („containing match“ im ersten, „contained match“ im zweiten Attribut), so überlappen sich die Anfragen gegenseitig („overlapping match“).

Auch die Komplexität dieses modifizierten Algorithmus ist, wie die des in Abschnitt 3.4 vorgestellten Verfahrens, nicht NP-vollständig. Die Einschränkung auf Konstantenselektionen führt hier zu einer maximal quadratischen Komplexität.

Der erhöhte Aufwand im Vergleich zum zuvor vorgestellten Algorithmus ist mit der Unterstützung des Operators '$\neq$' zu begründen. Bei Kollektion der angegebenen Filterwerte in einer Menge ist es innerhalb der Prozedur *build-ranges* nötig, diese Menge gegen die gerade betrachtete Teilbedingung zu prüfen. Dies kann im ungünstigsten Fall dazu führen, dass man in jedem Schritt über alle in der Menge enthaltenen Elemente iterieren muss und deren Anzahl der Anzahl bereits gelesener Teilziele entspricht. Allerdings führt eine einzige Bedingung mit dem '='-Operator schon zum Leeren der gesamten Menge $UnEq$, Eingrenzungen des gültigen Intervalls durch die Operatoren $\{<, \leq, >, \geq\}$ schränken sinnvolle Werte der Menge zumindest ein. Der Ablauf der inneren **for**-Schleife in Abbildung 14 entspricht im Wesentlichen dem Ablauf der Prozedur *build-ranges* für die Anfragen $q \wedge c$, $q \wedge \neg c$ und $c \wedge \neg q$. Der Test dieser drei Teile wird zur Bestimmung des *match-type* in einen einzigen Lauf über die *ranges* beider Anfragen vereint. Auch hier bestätigt sich die quadratische Komplexität als asymptotische obere Schranke, die unter Verwendung effizienterer Vorgehensweisen (z.B. mit Hilfe von assoziativen Speicherverfahren wie *Hash-Sets*) wohl auch noch verringert werden kann. Auf eine vertiefte Betrachtung und eventuelle Beweise wird hier abermals verzichtet, detaillierte Komplexitätsbetrachtungen könnten Inhalt späterer Arbeiten sein.

Beispiel

Es folgt ein kurzes Beispiel zur Veranschaulichung des *match-type*-Algorithmus für Zeichenketten. Angenommen, im Cache befindet sich die Anfrage c:

```
//Grafik[Kuenstler='van Gogh' and Motiv='Menschen']
```

mit den zugehörigen Ergebnisdaten. Nun wird die Anfrage q:

```
//Grafik[Kuenstler='van Gogh' and Datum='um 1600']
```

im Verlauf der Anfragebearbeitung an den Algorithmus übergeben. Die gebildeten *ranges* beider Anfragen sind (ohne Beschränkung der Allgemeinheit sei '0' das „kleinste" Zeichen des verwendeten Alphabets):

$c: range_{Kuenstler} =$ `( ['van Gogh','van Gogh0') , ∅)`
$range_{Motiv} =$ `( ['Menschen','Menschen0') , ∅)`

$q: range_{Kuenstler} =$ `( ['van Gogh','van Gogh0') , ∅)`
$range_{Datum} =$ `( ['um 1600','um 16000') , ∅)`

Auf den Aufruf von *match-type(q,c)* erhält man „overlapping" zurück. Die Gemeinsamkeit im Attribut Künstler wird erkannt, da aber Motiv in q nicht eingeschränkt wird und andererseits Datum nicht in c, erkennt der Algorithmus zum Einen im Attribut Datum q als die speziellere, zum anderen im Attribut Motiv als die allgemeinere der beiden Anfragen. Wie bei der Beschreibung des Algorithmus geschildert, entspricht der Wert beider Variablen *containing* und *contained* somit dem Wert WAHR und die Überlappung beider Anfragen wird korrekt erkannt.

Um die Behandlung des Ungleichheitsoperators zu veranschaulichen, wird eine weitere einfache Beispielanfrage und ihre Beziehung zur gespeicherten Anfrage c angegeben.

Angenommen, die neu ausgeführte Anfrage q ist:

```
//Grafik[Kuenstler≠'van Gogh' and Motiv='Menschen']
```

Die *ranges* der Anfrage haben dadurch folgende Form:

$$q : range_{Kuenstler} = (\ [-\infty, +\infty)\ ,\ \{\text{'van Gogh'}\})$$
$$range_{Motiv} = (\ [\text{'Menschen','Menschen0'})\ ,\ \emptyset)$$

Der Algorithmus erkennt in diesem Fall, dass der von c im Attribut Künstler festgelegte Wert 'van Gogh' in der Menge $UnEq_{Kuenstler}$ von q auftritt und gibt daraufhin „disjoint“ zurück.

5.2 Suche im Cache

In diesem Abschnitt wird der Algorithmus zur Suche nach passenden Einträgen im Cache formuliert. Er beschreibt das Vorgehen, zu einer gegebenen XPath-Anfrage q, die aus dem Cache erhältliche Antwort zu bestimmen, ist somit also das algorithmische „Herzstück“ des gesamten Caches und knüpft an die in Abschnitt 2.2 beschriebene Anfragebearbeitung an. Dabei wird zunächst nur der maximale schon im Cache gespeicherte Teil der Antwort bestimmt und parallel eine Anfrage nach dem verbleibenden, nicht gespeicherten (also bisher auch noch nicht angefragten) Teil, gebildet. Diese Komplementäranfrage kann, sofern sie nicht leer ist, nach Ablauf des Algorithmus dann an die enstprechenden Quellen gerichtet werden, während gleichzeitig die Anfrage an den physischen Speicher des Caches gerichtet wird. Es erfolgt also *zuerst* die Bestimmung der vorliegenden (Teil-)Antwort mit Hilfe der semantischen Daten und *anschließend* die eigentliche Anfrage nach den Ergebnis-Daten parallel auf Quelle und Cache, wenn nötig.

Zur Veranschaulichung des Ablaufs wird das in den letzten Abschnitten begonnene Beispiel fortgesetzt. Angenommen, der Cache befindet sich in einem frühen Zustand kurz nach seinem Start. Der einzige gespeicherte Eintrag ist die Anfrage

```
//Grafik[Kuenstler='van Gogh' and Motiv='Menschen']
```

mit dem zugehörigen Ergebnisdokument data.xml. Die neu zu bearbeitende Anfrage sei

```
//Grafik[(Kuenstler='van Gogh' or Kuenstler='Monet') and
                    Datum='um 1600']
```

Während der Suche im Cache wird jeder konjunktive Teilausdruck der disjunktiven Normalform der Anfrage q gegen alle Cache-Einträge geprüft, die bezüglich des in q angefragten Konzeptes gespeichert sind. Dabei findet kein Kriterium bezüglich der Auswahl der Reihenfolge der betrachteten Einträge Anwendung, obwohl Antwortzeit und Komplexität der resultierenden Komplementäranfrage davon offensichtlich beeinflusst werden können. Einige Überlegungen zur Bestimmung eines solchen Kriteriums werden bei der Behandlung der Verwaltung der semantischen Regionen (Abschnitt 4.3) und im Ausblick (Kapitel 7) genannt.

Die Korrektheit des bestimmten Ergebnisses wird durch die gewählte Reihenfolge nicht beeinflusst. Wie in Abschnitt 4.3 beschrieben, werden die im Cache gespeicherten Daten gegenseitig disjunkten semantischen Regionen zugeordnet. Unabhängig von der Reihenfolge der geprüften Cache-Einträge wird man also den gesamten im Cache vorliegenden Teil der Antwort finden, denn werden Daten einem bestimmten Cache-Eintrag zugeordnet, so sind diese Daten in keinem zweiten Eintrag zu finden. Existiert im Cache z.B. ein „exact match“ c, so wird man dies feststellen, egal welche und wieviele andere Einträge vor c geprüft werden (kein anderer Eintrag kann überhaupt einen Teil der Daten beinhalten).

Die disjunktive Normalform der zu bearbeitenden Anfrage im fortlaufenden Beispiel und somit die während der Suche im Cache bearbeitete Anfrage ist:

```
//Grafik[(Kuenstler='van Gogh' and Datum='um 1600')
    or (Kuenstler='Monet' and Datum='um 1600')]
```

Die beiden Teile in den runden Klammern stellen die beiden konjunktiven Ausdrücke dar, die im Verlauf des Algorithmus separat bearbeitet werden.

Abbildung 15 auf Seite 57 stellt den gesamten Ablauf der Prozedur *cache-lookup* dar.

Zu Beginn des Algorithmus wird durch den Aufruf von *disjunctive-normal-form(q)* (Zeile 1) die disjunktive Normalform (DNF) der Anfrage q in q' gebildet. Das Bilden der DNF, also einer zu q äquivalenten Anfrage, die aus nur disjunktiv verknüpften konjunktiven Ausdrücken besteht, ist zahlreich untersucht und beschrieben wurden. Der verwendete Algorithmus wird hier nicht näher erläutert, da er den bekannten Verfahren folgt.

Die Aufrufe von *get-concept* und *get-cache-entries* (beide Zeile 2) bestimmen das von q angefragte Konzept und die zu diesem Konzept im Cache gespeicherten Einträge. Die Einträge im Cache sind immer einem bestimmten Konzept zugeordnet. Eine Anfrage muss gegen die Teilmenge aller im Cache zum angefragten Konzept zugeordneten Einträge geprüft werden, nicht gegen alle. Die zum Prüfen aufgerufene Prozedur *match-type* (Zeile 10) entspricht dem in Abschnitt 5.1 beschriebenen Algorithmus zur Lösung des *Query Containment* für Zeichenketten. Hat man den *match-type* M bestimmt, so wird eine Referenz auf ein eventuell gespeichertes (Teil-)Ergebnis in R gespeichert und die weiterhin zu untersuchende Komplementäranfrage gebildet. $C \rightarrow Data$ symbolisiert einen Verweis auf die bezüglich des Cache-Eintrags C gespeicherten Daten. Der Verweis wird später benutzt, um die dem Anfrage-Ergebnis entsprechenden Daten aus dem Cache zu erhalten. Analog symbolisiert $Conj'(C \rightarrow Data)$, dass die gewünschten Daten dem Ergebnis der Anwendung des Prädikats $Conj'$ (ein konjunktiver Ausdruck) auf die zum Cache-Eintrag C zugehörigen Daten entsprechen.

Im Verlauf des Algorithmus muss man in jedem Schritt nur die aus vorigen Schritten resultierenden Teile der komplementären Anfrage betrachten. Wie bei der Beschreibung der Verwaltung der semantischen Regionen geschildert wurde, sind die in den Regionen erfassten Datenmengen gegenseitig disjunkt. Daten, die in einem Cache-Eintrag gefunden wurden, können in keinem anderen mehr enthalten sein. Die augenblicklich bearbeitete Anfrage filtert immer den noch verbleibenden, nicht bereits im Cache gefundenen, Teil der Ergebnisdaten.

Gegeben:
Query q
Rückgabe:
result set $R := \{\}$
complementary query $\overline{q} :=''$

```
1   q' = disjunctive-normal-form(q);
2   E = get-cache-entries (get-concept(q));
5   forall conjunction Conj of q' do
6     CC = {Conj};                    /* current conjunctions (still to check) */
7     forall cache entry C ∈ E do
8       NC := ∅;                      /* new conjunctions (to check) */
9       forall conjunctions Conj' ∈ CC do
10        M := match-type(Conj', C);
11        switch (M) do
12          case 'disjoint':    break;
13          case 'exact':       R := R ∪ C → Data;
14                              CC := CC \ {Conj'};
15                              break;
16          case 'containing':  R := R ∪ Conj'(C → Data);
17                              CC := CC \ {Conj'};
18                              break;
19          case 'contained':   R := R ∪ C → Data;
20                              CC := CC \ {Conj'};
21                              NC := NC ∪ {(Conj' ∧ ¬C)};
22                              break;
23          case 'overlapping': R := R ∪ Conj'(C → Data);
24                              CC := CC \ {Conj'};
25                              NC := NC ∪ {(Conj' ∧ ¬C)};
26                              break;
27        od
28      od
29      CC := CC ∪ NC;
30      if CC = {} then break;
31    od
32    if CC ≠ {} then q̄ := q̄ + CC;
33  od
34  return R, q̄;
```

Abbildung 15: Prozedur *cache-lookup*

Erfasst wird die aktuelle Anfrage durch die einzelnen konjunktiven Ausdrücke in CC. Nur die aus diesen Teilausdrücken gebildete Anfrage ist gegen weitere Cache-Einträge zu prüfen. Hat man also zumindest einen Teil des benötigten Ergebnisses gefunden, so kann man den gerade untersuchten Teilausdruck aus der Menge der noch zu untersuchenden Teilausdrücke entfernen (Zeilen 14, 17, 20 und 24). Handelt es sich nur um einen Teil der Antwortdaten, so müssen die restlichen Daten ebenfalls gesucht werden.

Um diese Daten korrekt zu beschreiben, ist die Bildung einer Komplementäranfrage nötig (Zeilen 21 und 25). Überlegungen und genaue Ausführungen dazu, hier einfach nur durch $Conj' \wedge \neg C$ beschrieben, sind in Abschnitt 5.3 zu finden. Wichtig ist hier zunächst, dass bei Bildung der Anfrage eine Menge neuer disjunktiv verknüpfter konjunktiver Ausdrücke entsteht, die gegen die verbleibenden Cache-Einträge geprüft werden müssen. Damit sie auch wirklich nur gegen die verbleibenden Einträge geprüft werden und nicht erneut gegen den gerade betrachteten, werden sie zunächst in der Menge NC gesammelt. Diese wird vor Prüfen eines jeden Cache-Eintrags mit der leeren Menge initialisiert (Zeile 8) und nach dem Abarbeiten von CC (den aktuell zu prüfenden konjunktiven Ausdrücken) schließlich neu zu CC hinzugefügt (Zeil 29). Dabei kann die gesamte Betrachtung beendet werden, d.h. eventuell verbleibende Cache-Einträge müssen nicht untersucht werden, wenn die Menge CC nach diesem Schritt leer bleibt (Zeile 30).

Die Bildung der globalen komplementären Anfrage (Zeile 32), also der Anfrage, die nach Beenden der Suche alle nicht im Cache gespeicherten Daten beschreibt und von der Prozedur zurückgegeben wird, ist ebenfalls vereinfacht beschrieben: $\overline{q} := \overline{q} + CC$ realisiert eine Verknüpfung der bisherigen (eventuell auch noch leeren) Komplementäranfrage $\overline{q}$ mit dem verbleibenden Rest der gerade bearbeiteten (Teil)-Anfrage, repräsentiert durch die in CC enthaltenen konjunktiven Ausdrücke, durch ein logisches „Oder“ ($\vee$). Sollte eine solche, nicht-leere Komplementäranfrage resultieren und zurückgegeben werden, so müssen nach Beantwortung dieser durch die Quellsysteme die erhaltenen Daten in den Cache integriert und die semantischen Regionen entsprechend aktualisiert werden. Näheres dazu findet sich in den Abschnitten 4.3 und 5.3.

Nachdem alle in der DNF von q enthaltenen Teilausdrücke und zusätzlich die durch Bilden der Komplementäranfrage neu entstehenden gegen alle Cache-Einträge geprüft wurden, werden die in R gesammelten Ergebnis-Verweise und die in $\overline{q}$ gebildete globale Komplementäranfrage an den Aufrufer zurückgegeben.

Im Gegensatz zu einigen anderen Arbeiten wird hier auch der wirklich maximale Teil der Antwort bestimmt, der schon im Cache gespeichert vorliegt. Eine Komplementäranfrage resultiert nur, wenn die bearbeitete Anfrage gegen *alle* Cache-Einträge geprüft wurde und trotzdem noch Daten fehlen. Ein komplettes Ergebnis kann somit auch dann erhalten werden, wenn kein „exact“ oder „containing match“ im Cache zu finden ist, z.B. durch Kombination mehrerer „contained matches“. Dass dieses Vorgehen eventuell kostenaufwendiger ist, als den Cache-Eintrag zu bestimmen, der den größten Teil der Antwort liefern kann und daraufhin den verbleibenden Rest gleich bei den Quellsystemen anzufragen, wird zugunsten der minimierten Netzauslastung und der maximalen erhaltenen Ergebnismenge gern in Kauf genommen.

In dem einfachen Beispiel wird zunächst der erste konjunktive Teilausdruck $conj_1$:

```
Kuenstler='van Gogh' and Datum='um 1600'
```

gegen alle gespeicherten Cache-Einträge geprüft. Dies ist im Beispiel nur ein einziger Eintrag zum Konzept `Grafik`, dessen Ergebnismenge mit der $conj_1$ entsprechenden Ergebnismenge überlappt. Auf den Aufruf der Prozedur *match-type* wird man in diesem Fall also ein „overlapping match“ erhalten.

Die demzufolge resultierende Komplementäranfrage ist:

```
//Grafik[Kuenstler='van Gogh' and Datum='um 1600' and
                    Motiv≠'Menschen']
```

Nach Beendigung der Suche im Cache wird $conj_1$ als Filterprädikat auf das zum Eintrag gespeicherte Ergebnisdokument data.xml angewandt und das erhaltene Ergebnis zum globalen Gesamtergebnis hinzugefügt. Da im Cache keine weiteren Einträge zum Konzept `Grafik` existieren, ist die Suche für $conj_1$ beendet.

Der verbleibende zweite Teilausdruck $conj_2$:

```
Kuenstler='Monet' and Datum='um 1600'
```

wird wiederum gegen den einzigen existierenden Cache-Eintrag geprüft. In diesem Fall stößt man auf ein „disjoint match“, da die beiden Ergebnismengen gegenseitig disjunkt sind. Dies ist daran ersichtlich, dass beide Anfragen zusammen im Attribut Künstler unerfüllbar sind. $conj_2$ wird somit unverändert zur globalen Komplementäranfrage hinzugefügt, diese hat daraufhin folgende Form:

```
//Grafik[(Kuenstler='van Gogh' and Datum='um 1600' and
                    Motiv≠'Menschen')
      or (Kuenstler='Monet' and Datum='um 1600')]
```

Der im Cache vorliegende Teil des Ergebnisses kann durch die XPath-Anfrage

```
//Grafik[Kuenstler='van Gogh' and Datum='um 1600']
```

aus dem physischen Speicher extrahiert werden. Dazu wird die XPath-Anfragekomponente der verwendeten XML-Datenbank, die in Kapitel 6 vorgestellt wird, genutzt. Die Komplementäranfrage und die Referenz zum Ergebnis in `data.xml` mit der entsprechenden Filteranfrage werden von der Prozedur *cache-lookup* an den Aufrufer zurückgegeben.

Das Laufzeitverhalten und die Kosten für eine Suche im Cache sind maßgeblich abhängig von der Anzahl an Einträgen, die bei einer Suche überprüft werden müssen. Dabei ist diese zwar wiederum abhängig von der Gesamtzahl im Cache gespeicherter Regionen, ist aber in der Regel größer als diese. Die verschiedenen Cache-Einträge werden für einzelne Teilziele der gerade untersuchten Anfrage mehrmals betrachtet. Die Anzahl auftretender Beziehungen (*match-types*) zwischen den Teilanfragen und den Cache-Einträgen ist hierbei ein wesentlicher Faktor. Die theoretische asymptotische Komplexität wird nicht näher betrachtet. Sie ergibt sich aus der Komplexität des *match-type*-Algorithmus zur Lösung der Frage des *Query Containment* in Verbindung mit der Anzahl möglicher Einträge im Cache. Die Effizienz des Algorithmus und die entstehenden Kosten müssen sich im Alltagstest mit entsprechendem Nutzerverhalten bewähren. Sie beruhen außer auf den Kosten der vorgestellten Algorithmen unter anderem auch auf der verwendeten Datenbank zur physischen Speicherung und den physischen Hardware-Voraussetzungen des genutzten Rechners. Erste Tests werden bereits in dieser Arbeit in Kapitel 6 ausgeführt und ausgewertet. Dort werden das Verhalten des Caches im Zeitablauf und der Aufwand zur Suche in Abhängigkeit von den erwähnten Faktoren näher betrachtet.

5.3 Bildung der Komplementäranfrage

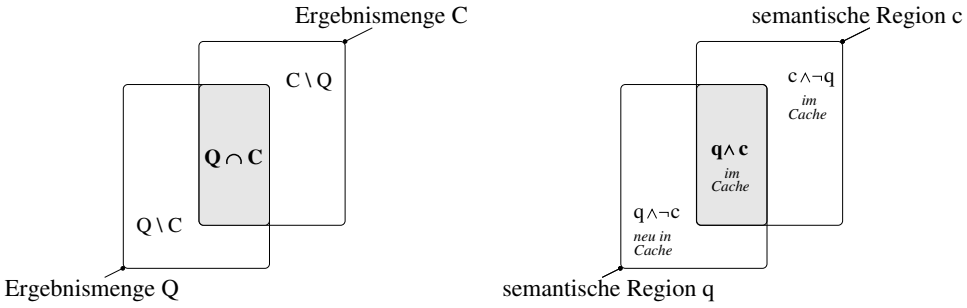
Eine Komplementäranfrage wird immer gebildet, wenn im Cache nur ein Teil der Ergebnisdaten gespeichert ist. Der verbleibende komplementäre Teil der Ergebnismenge kann durch Beantwortung der neu gebildeten Anfrage von den Quellsystemen erhalten werden. Die Komplementäranfrage entspricht im Cache somit immer der semantischen Beschreibung der fehlenden Ergebnisdaten.

Eine Komplementäranfrage entsteht bei der vorgestellten Anfragebearbeitung an zwei Punkten:

- Zum Einen entstehen implizite Komplementäranfragen durch die Bildung von konzeptbezogenen Teilanfragen im Zuge des *Query Containment* für Konzepte. Dies beruht auf der zu Grunde liegenden Hierarchie der Konzepte. Zur Erinnerung noch einmal das Beispiel aus Abschnitt 3.3: Wird in einer Anfrage q auf die Extension **ext**(c) mit zwei Unter-Konzepten c_1 und c_2, wobei gilt **ext**$(c) =$ **ext**$(c_1) \cup$ **ext**(c_2), Bezug genommen, so werden zwei Teilanfragen q_1 (nach **ext**(c_1)) und q_2 (nach **ext**(c_2)) gebildet und bearbeitet. Liegt **ext**(c_1) nun schon im Cache gespeichert vor, so kann q_1 aus dem Cache beantwortet werden und nur die Beantwortung von q_2 erfordert Kommunikation mit den Quellsystemen.

  q_2 stellt in diesem Fall somit die Komplementäranfrage zu q dar, es gilt $q_2 = \overline{q}$. Diese Bildung einer Komplementäranfrage ist wiederum Teil der Anfragebearbeitung außerhalb des Caches und wird hier nicht vertieft betrachtet.

- Eine Komplementäranfrage muss ebenfalls gebildet werden, wenn nur eine Teilmenge von **ext**(c_1) im Cache beschränkt durch ein Filterprädikat p gespeichert ist. In diesem Fall muss $\overline{q_1}$ durch Negierung von p schon während der Suche im Cache gebildet werden. Daher wird auf die Bildung einer solchen Anfrage im Folgenden näher eingegangen.



(a) Ergebnismengen

(b) semantische Regionen

Abbildung 16: Beziehung zwischen Ergebnismengen und semantischen Regionen

Angenommen, der schon im Cache vorliegende Teil des angefragten Gesamtergebnisses Q (zur Anfrage q) ist die gespeicherte Ergebnismenge C (zur Anfrage c).

Der Teil der Daten, der nicht im Cache gespeichert ist und durch die Komplementäranfrage bestimmt werden soll, ist die Differenz beider Mengen $Q \setminus C$. Die entsprechende Anfrage hat die Form $q \wedge \neg c$. Abbildung 16 verdeutlicht den Zusammenhang zwischen zwei Ergebnismengen und den entsprechenden semantischen Regionen noch einmal anschaulich. Das einer Region zugeordnete Prädikat beschreibt die in der Region enthaltenen Daten.

Da man sich in q und c auf jeweils nur konjunktiv verknüpfte Konstantenselektionen beschränkt, kann die resultierende Komplementäranfrage nach einem einfachen Verfahren bestimmt werden. Sei p das Prädikat in q, b das in c formulierte. Es gilt:

$$p \wedge \neg b = p \wedge \neg(b_1 \wedge \cdots \wedge b_l),$$

wobei l die Anzahl von einzelnen Teilbedingungen in b und $b_1 \ldots b_l$ diese Teilbedingungen darstellen. Aus den Gesetzen der Booleschen Logik folgt nun:

$$p \wedge \neg(b_1 \wedge \cdots \wedge b_l) = (p \wedge \neg b_1) \vee \cdots \vee (p \wedge \neg b_l).$$

Das Prädikat p der Anfrage q wird also mit jedem einzeln negierten Teilziel von b konjunktiv verknüpft, die entstehenden neuen konjunktiven Ausdrücke dann schließlich disjunktiv miteinander verbunden. Das Filterprädikat der resultierenden Anfrage enthält l untereinander disjunktiv verknüpfte konjunktive Ausdrücke, die wiederum jeweils $k+1$ konjunktiv verknüpfte Teilziele enthalten (k: Anzahl Teilbedingungen in p). Dabei entstehen auch einige Teilausdrücke, die unerfüllbar sind und somit aus der weiteren Betrachtung ausgeschlossen werden können. Ein kurzes Beispiel soll das verdeutlichen:

- Anfrage q:

  ```
  //Grafik[Kuenstler='van Gogh' and Motiv='Stillleben']
  ```

- gespeicherte Anfrage c:

  ```
  //Grafik[Kuenstler='van Gogh' and Datum='um 1600']
  ```

- $\rightarrow$ „overlapping match“

$\Rightarrow$ Komplementäranfrage:

```
//Grafik[(Kuenstler='van Gogh' and Motiv='Stillleben' and
                Kuenstler≠'van Gogh') or
(Kuenstler='van Gogh' and Motiv='Stillleben' and Datum≠'um
                          1600')]
```

Es ist sofort ersichtlich, dass der erste Teil der abgebildeten Komplementäranfrage unerfüllbar ist, denn es kann kein Datum geben, welches die Bedingungen 'Kuenstler='van Gogh" und 'Kuenstler≠'van Gogh" zugleich erfüllt. Weitere Betrachtungen können in diesem Fall also allein auf den zweiten Teil der Komplementäranfrage reduziert werden. Der Test auf Erfüllbarkeit (*satisfiability*), wie er in der Funktion *build-ranges* bei den *match-type*-Algorithmen in den Abschnitten 3.4 und 5.1 verwendet wird, findet hier ebenfalls Anwendung, um solche überflüssigen Teile der Anfrage zu erkennen und zu löschen.

Die Negierung von Operatoren aus der Menge $\{<, \leq, >, \geq\}$ und die Kombination entsprechender Teilziele mit der bearbeiteten Anfrage macht zusätzliche Umformungen möglich, die zur Vereinfachung der resultierenden Komplementäranfrage genutzt werden. So werden z.B. entstehende Intervalle zusammengefasst und Ungleichheitsbedingungen gelöscht, wenn der angegebene Wert nicht im zulässigen Intervall liegt. Wie schon zuvor beschrieben, könnte man auf ähnliche Weise auch alle Ungleichheitsbedingungen in der Anfrage „beseitigen“, indem man eine Teilbedingung '$A \neq c$' umformt zur Bedingung '$A < c \vee A > c$' und die dementsprechende disjunktive Normalform bildet. Beim Bilden der Komplementäranfrage wird diese Möglichkeit aber auch nicht näher betrachtet, da die einzelnen konjunktiven Teilausdrücke der resultierenden disjunktiven Normalform im weiteren Verlauf getrennt behandelt werden müssten und dies zu einem unnötigen Mehraufwand in der Anfragebearbeitung führen würde.

Durch die Form von p und b (nur Konjunktionen) ist es garantiert, dass die entstehende Komplementäranfrage nur konjunktive Teilausdrücke enthält, die alle disjunktiv verbunden sind. Allerdings erhält man nicht die disjunktive Normalform, denn nicht jeder Teilausdruck beschränkt auch alle möglichen Attribute, die Teilausdrücke stellen somit keine *Minterme*[3] der zu bildenden Anfrage dar. Die resultierende Anfrage ist korrekt und aufgrund ihrer Form ist die Bearbeitung im weiteren Verlauf möglich. Allerdings ist die vorliegende Form für die weitere Behandlung nicht optimal. Das Problem ist, dass die einzelnen konjunktiven Teilausdrücke sich gegenseitig überlappende Ergebnismengen beschreiben. Werden die Teilziele des gefundenen Cache-Eintrages einzeln negiert und mit der bearbeiteten Anfrage kombiniert, so entstehen Teilanfragen, die auch nur die auf die einzelnen Teilziele bezogenen komplementären Attributbelegungen beschreiben. Die Beziehung der Teilziele des Cache-Eintrages untereinander findet jedoch keine Berücksichtigung, wie es z.B bei Nutzung der disjunktiven Normalform der Fall wäre. Das Problem lässt sich wiederum am besten an einem einfachen Beispiel veranschaulichen:

Die gerade bearbeitete Anfrage q habe die Form:

```
//Grafik[Kuenstler='Gogh']
```

Der gerade bei der Suche im Cache gefundene Cache-Eintrag c sei:

```
//Grafik[Kuenstler='Gogh' and Motiv='Blumen' and Typ='Malerei'
                    and Datum='1600']
```

Offensichtlich ist die zu c im Cache gespeicherte Ergebnismenge eine Teilmenge der angefragten, denn c schränkt die selektierten Daten im Vergleich zu q durch drei zusätzliche Bedingungen ein. Eine nach beschriebenem Verfahren gebildete Komplementäranfrage hätte nun folgende Form:

```
//Grafik[(Kuenstler='Gogh' and Kuenstler≠'Gogh') or
     (Kuenstler='Gogh' and Motiv≠'Blumen') or
      (Kuenstler='Gogh' and Typ≠'Malerei') or
        (Kuenstler='Gogh' and Datum≠'1600')]
```

[3] Minterm steht hier für die konjunktive Verknüpfung *aller* in einem Prädikat vorkommenden Bedingungen. Jede Bedingung wird entweder negiert oder in ihrer Ursprungsform einbezogen. Die disjunktive Verknüpfung aller möglichen Minterme beschreibt das Prädikat und bildet die disjunktive Normalform.

Der erste der vier Teilausdrücke ist im Attribut Künstler nicht erfüllbar und wird ignoriert. Es ist leicht zu zeigen, dass die durch die verbleibenden Teilausdrücke beschriebenen Ergebnismengen sich überlappen. So ist z.B. ein Objekt mit den Attributbelegungen

```
Kuenstler='Gogh', Motiv='Stillleben', Typ='Gemälde',
                  Datum='1700'
```

in den Ergebnismengen aller drei Teilausdrücke enthalten. Ein solches Objekt sollte aber nur einer weiterhin bearbeiteten Anfrage zugeordnet werden, denn die Teilanfragen beschreiben die fehlenden Daten und am Ende der Suche schließlich die neu zu bildenden semantischen Regionen. Werden die verbleibenden Teile also im weiteren Verlauf der Suche im Cache in dieser Form behandelt, so bilden sie unter Umständen, eventuell nach weiteren Transformationen im Zuge der Bildung einer komplementären Anfrage, einen Teil der globalen Komplementäranfrage am Ende der Suche.

Abbildung 17 verdeutlicht den Zusammenhang beider Anfragen. Die nur angedeuteten Regionen symbolisieren die Ergebnismengen, die der Einschränkung in nur einem der drei zusätzlichen Attribute entsprechen würden. Der gemeinsame Teil dieser drei, nicht im Cache vorliegenden, semantischen Regionen, ist der Cache-Eintrag c. Sie sind hier angedeutet, um die sieben zu unterscheidenden Teilmengen (markiert durch die Zahlen 1 bis 7) der zur Datenmenge von c komplementären Ergebnismenge bezüglich q unterscheiden zu können.

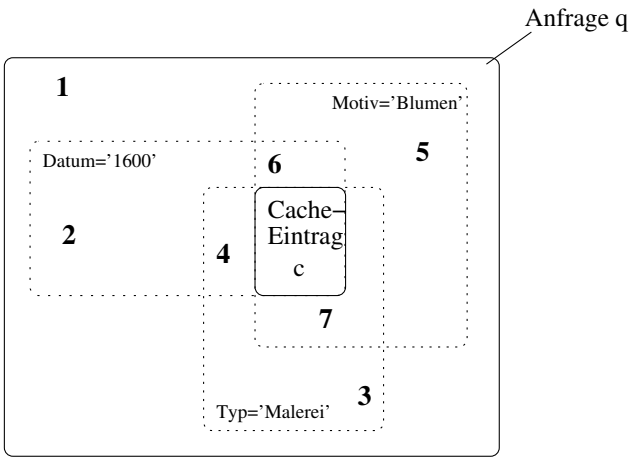


Abbildung 17: Zu Cache-Eintrag c komplementäre Ergebnismenge der Anfrage q

Eine Möglichkeit ist, die resultierenden Teilanfragen vor weiterer Bearbeitung oder spätestens nach Bildung der globalen Komplementäranfrage, auf jeden Fall aber vor Speicherung erhaltener Ergebnisdaten im Cache, gegeneinander auf Überlappungen zu prüfen und inklusive der Ergebnismengen entsprechend zu modifizieren. Dazu könnten die vorgestellten Methoden der Anfragebearbeitung genutzt werden. Dies würde sich natürlich in einem erhöhten Bearbeitungsaufwand zur Bildung der Komplementäranfrage oder aber im Lauf der weiteren Anfragebearbeitung bemerkbar machen.

Anfrage	Künstler	Motiv	Typ	Malerei	Teilmenge in Abb. 17
$\overline{q}_1$	=’Gogh’	$\neq$’Blumen’	$\neq$’Malerei’	$\neq$’1600’	1
$\overline{q}_2$	=’Gogh’	$\neq$’Blumen’	$\neq$’Malerei’	=’1600’	2
$\overline{q}_3$	=’Gogh’	$\neq$’Blumen’	=’Malerei’	$\neq$’1600’	3
$\overline{q}_4$	=’Gogh’	$\neq$’Blumen’	=’Malerei’	=’1600’	4
$\overline{q}_5$	=’Gogh’	=’Blumen’	$\neq$’Malerei’	$\neq$’1600’	5
$\overline{q}_6$	=’Gogh’	=’Blumen’	$\neq$’Malerei’	=’1600’	6
$\overline{q}_7$	=’Gogh’	=’Blumen’	=’Malerei’	$\neq$’1600’	7
$\overline{q}_8$	=’Gogh’	=’Blumen’	=’Malerei’	=’1600’	c

Tabelle 3: Alle Minterme der resultierenden Komplementäranfrage

Werden die resultierenden Teile gleich mit Aufrufen der Prozedur *match-type* geprüft und anschließend modifiziert, so steigt der Aufwand zur Bildung der komplementären Anfrage der in jedem Schritt anfällt. Lässt man die im weiteren Verlauf bearbeiteten Teilanfragen sich gegenseitig überlappen, erhöht das den Aufwand zur Suche, denn dadurch werden während der Anfragebearbeitung bereits im Cache gespeicherte Daten mehrfach als Teil der Ergebnismenge erfasst.

Alternativ könnte an diesem Punkt die disjunktive Normalform der resultierenden Anfrage gebildet werden, denn dann würden die Teilausdrücke sich nicht mehr gegenseitig überlappen. Eine Bildung in zwei Schritten, d.h. zuerst eine komplementäre Anfrage nach erläutertem Verfahren zu erstellen und dann die entsprechende Normalform zu bilden, ist allerdings nicht nötig. Schon während der Bestimmung einer komplementären Anfrage kann man darauf achten, diese in disjunktiver Normalform zu erstellen. Dazu könnte man, ähnlich dem Aufbau einer Wahrheitswerttabelle, alle möglichen Beziehungen zwischen den Teilzielen des Cache-Eintrages bilden. Dabei besteht für jedes Teilziel die Wahl, ob es in die resultierende Anfrage negiert einbezogen wird oder nicht negiert. Dies entspricht der Bestimmung der Minterme der Komplementäranfrage. Tabelle 3 fasst alle Teilanfragen zusammen, die auf diese Weise im angegebenen Beispiel entstehen.

Das Problem ist die hohe Anzahl von resultierenden Teilanfragen, denn jede der in Abbildung 17 einzeln markierten Teilmengen wird durch eine einzelne Anfrage beschrieben. Dies sind in der Regel nicht alle der $2^{|c|}$ ($|c|$: Anzahl Teilziele im Cache-Eintrag c) möglichen Anfragen, denn einige ergeben sich als nicht erfüllbar und können ignoriert werden. Zu ignorieren ist hier auf jeden Fall die letzte der gebildeten Teilanfragen, denn sie entspricht dem Cache-Eintrag c. Die Anzahl resultierender Anfragen und damit der Aufwand in der folgenden Anfragebearbeitung lässt sich merklich reduzieren, wendet man das Wissen über die abgebildeten Teilmengen an und modifiziert das vorgestellte Verfahren. Dabei fasst man verschiedene Teilmengen der in Abbildung 17 dargestellten zusammen und bestimmt Teilausdrücke, die die jeweilige Kombination beschreiben. Die einzelnen Teilziele werden wiederum nacheinander abgearbeitet, die Beziehungen zwischen den Teilzielen in c werden berücksichtigt: Zunächst wird das betrachtete Teilziel negiert und mit q kombiniert, daraufhin wird die entstehende Teilanfrage mit allen zuvor betrachteten Teilzielen in ihrer ursprünglichen (nicht negierten) Form verknüpft.

Das Vorgehen und die Form der resultierenden Anfrage, mit Zuordnung zu den Teilmengen aus Abbildung 17, wird verdeutlicht am laufenden Beispiel:

```
                                //Grafik[

(Kuenstler='Gogh' and Motiv≠'Blumen') or    ≙ {1 ∪ 2 ∪ 3 ∪ 4}
(Kuenstler='Gogh' and Typ≠'Malerei'
                  and Motiv='Blumen') or    ≙ {5 ∪ 6}
(Kuenstler='Gogh' and Datum≠'1600'
                  and Motiv='Blumen'
                  and Typ='Malerei')        ≙ {7}

                                ]
```

Man kann unter den resultierenden Teile der Komplementäranfrage eine gewisse „Verwandtschaft“ erkennen. Es bildet sich eine Art Hierarchie, denn da zuletzt gebildete Teile Bezug auf die vorher gebildeten nehmen, sind sie als spezieller anzusehen. Alle Teilanfragen beziehen sich auf den gleichen Teil der Ergebnismenge, den zu c komplementären Teil der Ergebnismenge von q, beschreiben aber verschiedene Teile dieser Daten. Die Teilausdrücke entsprechen in ihrer Form den Teilanfragen, die bei Anwendung des am Anfang dieses Abschnitts beschriebenen Verfahrens und anschließender Modifikation mit den Methoden des Caches entstehen würden. Die resultierende Anfrage ist nicht in disjunktiver Normalform, aber die enthaltenen Teilausdrücke beschreiben auch keine sich überlappenden Datenmengen und sind somit für die weitere Bearbeitung gut geeignet. Zudem ist der zu erwartende Aufwand geringer als bei Nutzung der disjunktiven Normalform, da weniger Teilanfragen entstehen.

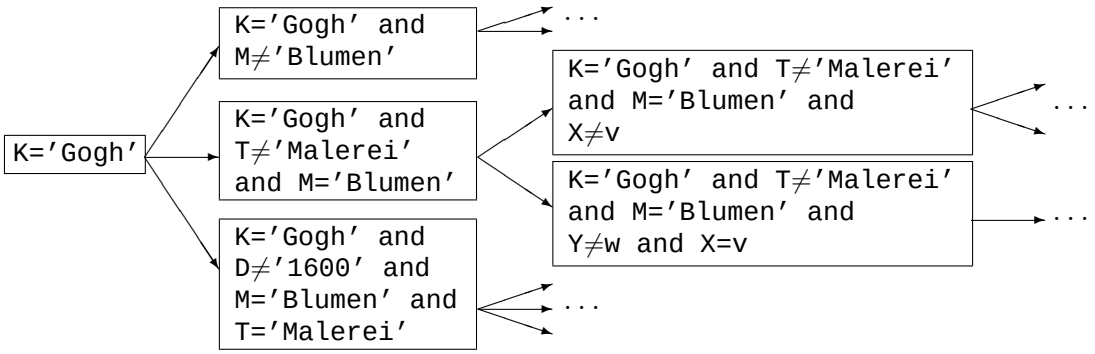


Abbildung 18: Schrittweise Bildung der globalen Komplementäranfrage

Die Entstehung der globalen Komplementäranfrage lässt sich veranschaulichen, indem man die einzelnen Schritte auf dem Weg ihrer Bildung in einer Art Baumgrafik zusammenfasst. Ein Schritt entspricht dabei der Bildung einer komplementären (Teil-)Anfrage nach dem soeben beschriebenen Verfahren, wenn Überlappungen zwischen einer gerade bearbeiteten Anfrage und einem Cache-Eintrag vorliegen. In Abbildung 18 ist ein Teil dieser Baumgrafik für das angegebene Beispiel dargestellt. Die Buchstaben `K`, `M`, `T` und `D` werden als Synonyme für die Attribute Künstler, Motiv, Typ und Datum verwendet.

X und Y stehen für zwei Attribute, die in einem weiteren überlappenden Cache-Eintrag c' mit '$=$' auf v bzw. w eingeschränkt werden. X und Y können hier einem oder zwei verschiedenen Attributen der Menge {K, M, T, D} entsprechen, aber auch ein oder zwei nicht in dieser Menge enthaltenen Attributen. Für v und w folgt nur, dass sie abhängig von X und Y keine Werte annehmen können, die c' oder die gebildete Komplementäranfrage nicht erfüllbar machen. Entspricht X oder Y einem bereits eingeschränkten Attribut, so kann sich die entstehende Anfrage auch verkürzen, indem Teilbedingungen vereint werden. In diesem Fall kann das nur passieren, wenn v bzw. w ebenfalls einem zuvor angegebenem Attributwert entspricht. Bei Nutzung anderer Operatoren könnten z.B. Intervalle zusammengefasst werden. Die nicht abgebildeten Teile der Grafik, symbolisiert durch '. . . ', führen nicht immer zu resultierenden Teilen der globalen Komplementäranfrage. Wird in einem Schritt das einer bearbeiteten Teilanfrage entsprechende komplette Ergebnis im Cache gefunden, so endet die Suche in diesem Punkt und keine weiteren Teilausdrücke werden aus dieser Stufe zu der globalen Komplementäranfrage hinzugefügt.

Je mehr passende Einträge bei der Suche im Cache gefunden werden, die mit der gerade betrachteten Anfrage überlappen, desto komplexer wird die im weiteren Verlauf zu bearbeitende Anfrage. Dies folgt daraus, dass in jedem Schritt, in dem ein solcher Eintrag gefunden wird, eine Komplementäranfrage nach oben geschildertem Verfahren gebildet wird, welche die im weiteren Verlauf zu bearbeitende Anfrage darstellt. Die einzelnen Komplementäranfragen, die am Ende der Suche resultieren, werden disjunktiv miteinander verbunden und als globale Komplementäranfrage von der Cache-Verwaltung zurückgegeben.

Jeder Eintrag, der einen Teil der Ergebnisdaten enthält, kann die Anfrage also um einige Teilausdrücke erweitern. Dadurch kann die resultierende und im weiteren Verlauf zu bearbeitende Anfrage sehr komplex werden. Solch komplexe Anfragen können unter Umständen, eventuell beim Übersetzen in die Quellformate noch in viele kleinere Anfragen geteilt, einen erheblichen Mehraufwand bei Beantwortung verursachen oder überhaupt nicht beantwortbar sein (z.B. durch nicht unterstützte Operatoren). Zur Bildung einer Komplementäranfrage müssen die Operatoren der einzelnen Teilbedingungen einer Anfrage negiert werden. Die Unterstützung einer solchen allgemeinen Negierung kann aber nicht von jeder zu integrierenden Quelle erwartet werden, ohne die Skalierbarkeit und Flexibilität des Systems merklich einzuschränken. Besondere Probleme ergeben sich durch Unterstützung des Textähnlichkeitsoperators '$\sim=$'. Abschnitt 5.4 beschäftigt sich mit den resultierenden Problemen und ihrer Lösung.

Es werden also Heuristiken benötigt, nach denen man entscheiden kann, ob eine resultierende Komplementäranfrage im weiteren Verlauf bearbeitet werden sollte oder lieber die Originalanfrage. Wie sich herausstellen wird, ist es für Anfragen, die den '$\sim=$'-Operator enthalten naheliegend, statt den resultierenden Komplementäranfragen immer die Originalanfragen an die Quellen zu senden. Um Anfragen bewerten zu können, die nur andere Operatoren beinhalten, sind Informationen über Quelleneigenschaften wünschenswert. Wird die Originalanfrage anstatt der komplementären zu den Quellen gesandt, so ist zum Einen eine erhöhte Netzlast zu erwarten und zum anderen eine Duplikateliminierung notwendig (siehe dazu Abschnitt 5.5).

Trotz des erhöhten Aufwands unterstützt der Cache die schnellere Generierung und Übermittlung eines Teilergebnisses an den Nutzer: im Cache vorliegende Daten können, ohne auf eine Beantwortung durch die Quellsysteme warten zu müssen, schon zurückgegeben werden. Nur das wirklich vollständige Ergebnis kann erst nach Ausführen der Quellanfragen bestimmt und übermittelt werden. Mit Behandlung des '$\sim=$'-Operators wird in Abschnitt 5.4 ebenfalls die im Cache umgesetzte Vorgehensweise geschildert.

Die Bildung einer Komplementäranfrage während der Suche im Cache ist allerdings zwingend nötig. Um die im Cache erfassten Daten im Laufe der Anfragebearbeitung und des realisierten Ersetzungsverfahrens korrekt behandeln zu können, müssen die den semantischen Regionen zugeordneten konjunktiven Ausdrücke auch wirklich die Daten beschreiben, die in den Regionen tatsächlich erfasst sind. Das ist z.B. Voraussetzung für die Unabhängigkeit von der Reihenfolge der untersuchten Einträge während der Suche im Cache. Da die Regionen untereinander disjunkt sind, dürfen auch die zugeordneten Prädikate keine sich überlappenden Ergebnismengen beschreiben. Egal, ob nun also die komplementäre Anfrage oder stattdessen die Originalanfrage zur Beantwortung an die Quellsysteme geleitet wird, die gebildete Komplementäranfrage beschreibt die neu im Cache zu erfassenden Daten und muss deshalb bestimmt werden.

5.4 Unterstützung des Textähnlichkeitsoperators

Spezielles Augenmerk bei der Anfragebearbeitung im Cache ist CQuerys Textähnlichkeitsoperator '$\sim=$' zu widmen. Durch ihn werden unscharfe Anfragen nach Ähnlichkeiten in den Attributbelegungen möglich, allerdings ist diese Ähnlichkeit nicht eindeutig festgelegt. Der Operator wird auf entsprechende Operatoren der Quellsysteme abgebildet, wie z.B. like oder contains. Wie in Kapitel 2 schon erwähnt wurde, wäre eine erfolgreiche Suche ohne diesen Operator nur möglich, wenn man zumindest einige der möglichen Attributbelegungen exakt im Voraus angeben kann. Da diese aber im Allgemeinen nicht exakt bekannt sind, würde eine Suche äußerst langwierig und mit vielen leeren Resultatmengen verbunden sein.

Um während der Suche im Cache die Beziehung zwischen neuer und gespeicherter Anfrage bestimmen zu können, muss man in der Lage sein, solche Ähnlichkeiten in Attributbelegungen zu erkennen, allerdings ohne etwas über die Semantik der entsprechenden Operatoren und deren Realisierung in den Quellsystemen zu wissen. Ohne näheres Wissen über Semantik und Umsetzung ist eine korrekte Bestimmung von Überlappungen in Anfragen und die Bildung entsprechender Komplementäranfragen allerdings nicht möglich. Kann oder will man Informationen über Quelleneigenschaften nicht einbeziehen (in den meisten Fällen verfügt man über solche Informationen nicht) bietet sich zunächst die Möglichkeit, sich auf *eine* denkbare Realisierung des Operators zu beschränken und die Anfragebearbeitung entsprechend anzupassen.

Das bis zu diesem Punkt beschriebene Vorgehen während der Anfragebearbeitung ändert sich durch Integration des '$\sim=$'-Operators nicht wesentlich, weshalb die Modifikationen hier in einem eigenen Abschnitt behandelt werden. Die in den vergangenen Abschnitten erfassten Voraussetzungen treffen nun ebenfalls zu.

Die vorgestellten Algorithmen sind im Kern korrekt und notwendige Änderungen geringfügig. Ebenso können die zur Erläuterung aufgeführten Beispiele problemlos auf die hier beschriebene Problematik erweitert werden. Der nun folgende Ansatz ist nur ein möglicher Ansatz zur Lösung der Probleme mit dem Textähnlichkeitsoperator. Das realisierte Verfahren ist recht einfach und resultiert in einigen speziellen Problemen, die ebenfalls näher betrachtet werden. Andere denkbare Ansätze werden kurz in Kapitel 7 skizziert. Anfragen, die den Textähnlichkeitsoperator nicht enthalten, sind von den geschilderten Problemen und dem Großteil der im darauf folgenden Abschnitt behandelten Aspekte zur Speicherung komplementärer Daten nicht betroffen. Für solche Anfragen können die bis zu diesem Abschnitt vorgestellten Verfahren ohne besondere Anpassungen problemlos angewandt werden.

Die Lösung der Problematik basiert hier auf der Inklusion von Zeichenketten, also der Frage, ob sich zwei Zeichenketten gegenseitg enthalten. Wird in einer Anfrage der '$\sim=$'-Operator in einem Attribut genutzt, so wird das erhaltene Ergebnis alle Objekte enthalten, deren Belegung in diesem Attribut den Filterwert als Teil-Zeichenkette enthält. Ein Beispiel soll das verdeutlichen: ist ein einer Anfrage die Selektion `Artist` $\sim=$ `'Gogh'` angegeben, so sind in der Ergebnismenge Objekte mit Attributbelegungen wie `Artist = 'v. Gogh', Artist = 'van Gogh, V.', Artist= 'v. Gogh, V'` etc. enthalten. Wird in einer neuen Anfrage dann nach `Artist` $\sim=$ `'van Gogh'` gefiltert, so kann das gespeicherte Resultat unter Filterung durch die neue Anfrage verwendet werden, denn `'Gogh'` ist ein Teil der Zeichenkette `'van Gogh'`. Alle Objekte, die auf die neue Anfrage zurückgegeben werden, sind also schon in der Ergebnismenge der gespeicherten Anfrage enthalten. Allgemein kann man die umgesetzte Regel folgendermaßen formulieren:

Angenommen, zwei Anfragen filtern das gleiche Attribut A durch den '$\sim=$'-Operator. Sei die Bedingung der ersten Anfrage $A \sim= x$, die der zweiten $A \sim= y$. Ist x ein Teil der Zeichenkette y, so enthält ein Cache-Eintrag zu $A \sim= x$ auch alle Ergebnisobjekte zu $A \sim= y$. Wären andererseits Daten zu $A \sim= y$ gespeichert und $A \sim= x$ bildet die Filterbedingung der neuen Anfrage, so wäre nur eine Teilmenge der Ergebnismenge im Cache gespeichert („contained match“) und eine Komplementäranfrage müsste gebildet werden.

In Abschnitt 4.4 wurden bereits die Anforderungen an den physischen Speicher des Caches erfasst und die Bedeutung einer effizienten XPath-Anfrageunterstützung hervorgehoben. Die in der Implementierung zu treffende Auswahl wurde dort auf einen Vertreter der nativen XML-Datenbanken begrenzt, welche in der Regel XPath als Anfragesprache unterstützen. Die in der XPath-Spezifikation[4] festgelegte Funktion *contains* prüft für eine Zeichenkette, ob eine zweite Zeichenkette in ihr enthalten ist. Da diese Funktion laut Spezifikation von jeder XPath-Implementierung realisiert werden muss, ist eine Umsetzung des geschilderten Verfahrens zur Behandlung des '$\sim=$'-Operators problemlos möglich. Bei Bildung einer Komplementäranfrage müssen Teilziele, die ein '$\sim=$' enthalten, auch negiert werden (siehe dazu Abschnitt 5.3). Zur Anfrage auf dem physischen Speicher wird dazu die in XPath-Anfragen mögliche Negierung eines Teilzieles durch die Funktion *not* genutzt.

[4] siehe `http://www.w3.org/TR/xpath`

Das modifizierte algorithmische Vorgehen wird im Folgenden skizziert. Da die Änderungen bei der Suche im Cache und zur Bestimmung des *match-type* zwischen zwei Anfragen recht geringfügig sind, werden die Prozeduren hier nicht erneut im Pseudo-Code gelistet. Auf den in den Abschnitten 5.1 und 5.2 erläuterten Algorithmen basierend wird das angewandte Verfahren in Prosa beschrieben. Die folgenden Ausführungen fassen den Ablauf der bereits vorgestellten Methoden zudem noch einmal anschaulich zusammen.

Die Änderungen an der Prozedur *build-ranges* lassen sich einfach in einer kurzen Auflistung erfassen, zunächst aber noch ein paar einführende Worte: Wie bisher werden innerhalb der Prozedur die einzelnen Teilziele einer Anfrage gelesen und die *ranges* der Anfrage bezüglich den Operatoren angepasst, wobei ebenfalls eine Kontrolle auf die Erfüllbarkeit der Anfrage integriert ist. Die *ranges* umfassen nach wie vor die für die einzelnen Attribute gültigen Intervalle und die Mengen $UnEq$ zur Erfassung des Ungleichheitsoperators. Zusätzlich werden nun zwei Mengen Sim und $UnSim$ verwaltet, die analog zu $UnEq$ zur Erfassung von Bedingungen mit dem Operator '$\sim=$' dienen. Ein Einsatz von Bedingungen der Form '$\neg$(A $\sim=$ 'xxx')' (von nun an: 'A $\not\sim=$ 'xxx') ist zur Anfragebildung im YACOB-Mediator nicht vorgesehen. Solche „Nicht-Ähnlichkeiten" entstehen allerdings, im Übrigen analog zu Bedingungen mit '$\neq$', bei der Negierung von Teilzielen im Zuge der Bildung einer Komplementäranfrage und müssen erfasst werden. Sim als Menge zu realisieren wäre unnötig, würde man den Operator '$\sim=$' ähnlich dem Gleichheitsoperator '$=$' unterstützen. In diesem Fall würde ein einzelner Sim-Wert genügen, den ein selektiertes Objekt im entsprechenden Attribut enthalten muss. Allerdings ist eine Anfrage wie z.B.

```
//Grafik[Titel∼='Stillleben' and Titel∼='Äpfel']
```

durchaus von Nutzen, um alle Objekte mit Titeln wie „Stillleben aus Äpfeln", „Äpfel-Stillleben" usw. anzufragen, weshalb Kombinationen von Bedingungen mit '$\sim=$' hier auch für ein und dasselbe Attribut unterstützt werden.

Die folgende Auflistung ordnet den möglichen Operatoren die jeweilige Aktion innerhalb der Prozedur zu. Dabei wird im Wesentlichen Bezug auf auf die Mengen Sim und $UnSim$ sowie deren Einfluss auf die anderen Elemente einer *range* genommen. Die Grenzen der gültigen Intervalle einer *range*, $[min, max)$, sind von Bedingungen mit dem Operator '$\sim=$' im Allgemeinen nicht betroffen, ebenso die neu eingeführten Mengen nicht von Operatoren aus der Menge $\{<, \leq, >, \geq\}$. Dies ist ganz einfach durch die zu Grunde liegende Semantik der Operatoren zu begründen: Operatoren wie '$<$' und '$>$' legen eine Ordnung auf Zeichenketten fest, die abhängig von der Reihenfolge der enthaltenen Zeichen ist. Der Textähnlichkeitsoperator (bei der hier interpretierten Semantik) beschränkt mögliche Werte auf diejenigen, die den angegeben Parameter als Teil enthalten. Da dieser Teil aber nun mitten im Wort, am Ende oder auch am Anfang auftauchen kann, allerdings dem Attributwert auch exakt entsprechen kann, ist eine Anpassung der gültigen Intervalle meistens einfach nicht möglich. Vielmehr wäre sie oft sogar falsch, da es sich nicht um eine Einschränkung bezüglich der Intervalle handelt. In anderer Richtung gilt dies ebenso: aus Worten, die in einem bestimmten Intervall liegen, kann man im Allgemeinen keinen Schluss auf enthaltene Zeichenketten ziehen. Möglich wird beides bei sehr speziellen Bedingungs-Kombinationen, deren Erkennung und Behandlung allerdings recht komplex wäre und die zudem recht selten auftreten.

gelesen	nicht erfüllbar wenn	Anpassungen *ranges*
A = c	$c < min$ oder $c \geq max$ $\exists x \in Sim : x \not\subseteq c$ $\exists y \in UnSim : y \subseteq c$ $c \in UnEq$	$min = c,\ max = c + 1$ $Sim = \emptyset$ $UnSim = \emptyset$ $UnEq = \emptyset$
A $\neq$ c	$min + 1 = max$ und $c = min$	$UnEq = UnEq \cup \{c\}$, aber nur wenn: $min + 1 \neq max$ und $c \geq min$ und $c < max$ und $\forall x \in Sim : x \subseteq c$ und $\forall y \in UnSim : y \not\subseteq c$
A $\sim=$ c	$min + 1 = max$ und $c \not\subseteq min$ $\exists y \in UnSim : y \subseteq c$	$Sim = Sim \cup \{c\}$, aber nur wenn: $min + 1 \neq max$ und $\forall x \in Sim : x \not\supseteq c$ $\forall x \in Sim : x \subset c \rightarrow Sim = Sim \setminus \{x\}$ $\forall z \in UnEq : z \not\supseteq c \rightarrow UnEq = UnEq \setminus \{z\}$
A $\not\sim=$ c	$min + 1 = max$ und $c \subseteq min$ $\exists x \in Sim : x \supseteq c$	$UnSim = UnSim \cup \{c\}$, aber nur wenn: $min + 1 \neq max$ und $\forall y \in UnSim : y \not\subseteq c$ $\forall y \in UnSim : y \supset c \rightarrow UnSim = UnSim \setminus \{y\}$ $\forall z \in UnEq : z \supseteq c \rightarrow UnEq = UnEq \setminus \{z\}$

Tabelle 4: Prozedur *build-ranges* mit Unterstützung von '$\sim=$'

Ein sehr einfaches Beispiel: Intervall zu Attribut $A : [min, max) = (\textit{'Gogh'}, \textit{'Goghaa'})$ $[(min + 1) + 1 = max!]$ und Bedingung 'A $\sim=$ 'Malchin'' → nicht erfüllbar.

Auf erneute Behandlung der Operatoren aus der Menge $\{<, \leq, >, \geq\}$ wird hier aufgrund ihrer geringen Bedeutung somit verzichtet. Nach wie vor aber gilt: $range_A : min + 1 = max$, also im gültigen Intervall $[min, max)$ zum Attribut A ist max der lexikographische Nachfolger (siehe Abschnitt 5.1) von min, genau dann wenn das untersuchte Prädikat der Bedingung 'A = min' entspricht. Bedingungen mit dem Gleichheitsoperator '$=$' haben natürlich Einfluss auf Sim und $UnSim$, wie sich zeigen wird.

Im Folgenden bestehe die gerade betrachtete *range* aus dem Intervall $[min, max)$ und den Mengen $Sim, UnSim, UnEq$. Tabelle 4 fasst die Behandlung der einzelnen Operatoren zusammen. Die Operatoren '$\subseteq$' und '$\supseteq$' repräsentieren die Teil-von-Beziehung auf Zeichenketten, d.h. '$x \subseteq y$' ist wahr, wenn x ein Teil der oder gleich der Zeichenkette y ist.

Aufbauend auf dem beschriebenen Ablauf der Prozedur *build-ranges* kann man für die durch die Prozedur gebildeten $ranges_A : ([min, max), Sim, UnSim, UnEq)$ folgende Regeln formulieren:

i) Wenn $min + 1 = max$ (d.h. Anfrage-Prädikat entspricht der Bedingung 'A = min'), dann gilt: $Sim = \emptyset, \quad UnSim = \emptyset, \quad UnEq = \emptyset$.

ii) Wenn $min = c$, dann gilt: $max > c$,
$$\forall z \in UnEq : z \geq c.$$

iii) Wenn $max = c$, dann gilt: $min < c$,
$\forall z \in UnEq : z < c$.

iv) Wenn $x \in Sim$, dann gilt: $min + 1 \neq max$,
$\forall x' \in Sim, x' \neq x : x' \not\supseteq x$,
$\forall z \in UnEq : z \supseteq x$.

v) Wenn $y \in UnSim$, dann gilt: $min + 1 \neq max$,
$\forall y' \in UnSim, y' \neq y : y' \not\subseteq y$,
$\forall z \in UnEq : z \not\supseteq y$.

vi) Wenn $z \in UnEq$, dann gilt: $\min \leq z$,
$max > z$,
$\forall x \in Sim : x \subseteq z$,
$\forall y \in UnSim : y \not\subseteq z$.

Die aufgeführten Regeln gelten alle gleichzeitig, keine wird durch eine andere bedingt. Anhand der Auflistung und Tabelle 4 wird eine weitere Modifikation des Vorgehens ersichtlich, die auch schon bei dem Algorithmus in Abschnitt 3.4 möglich gewesen wäre. Dort kann die Menge $UnEq$ überflüssige Einträge enthalten, d.h. Elemente, die außerhalb des angegebenen gültigen Intervalls liegen. Solche Elemente erhöhen den Aufwand bei Überprüfung der Menge $UnEq$ unnötig, das Vorgehen allerdings ist trotzdem korrekt. Im Algorithmus in Abschnitt 5.1 wurde dieser Punkt schon beachtet und integriert, allerdings ohne näher darauf einzugehen. Mit Unterstützung des Operators '$\sim=$' und seiner Negierung steigt die Anzahl solch überflüssiger Werte in einer *range*, denn nun können auch die Mengen Sim und $UnSim$ überflüssige Elemente enthalten. Die Lösung ist ein Vergleich der in den Mengen enthaltenen Zeichenketten nach dem Lesen einer Operation und Entfernung entsprechender überflüssiger Elemente. Dies wurde bei dem hier vorgestellten Verfahren umgesetzt. Wie das ungefähr geschieht und auf welchen Zusammenhängen das Vorgehen beruht, ist z.B. anhand der Regeln vi): „*kein* Element in $UnEq$, das außerhalb des Intervalls $[min, max)$ liegt, das *nicht auch* in allen Elementen der Menge Sim enthalten ist oder das *auch* ein Element der Menge $UnSim$ enthält“ oder aber v): „*kein Element* in $UnSim$, das *auch* ein anderes Element aus der Menge $UnSim$ enthält“, erkennbar. Diese Beziehungen der Operatoren müssen bei Bestimmung der Beziehung zwischen zwei Anfragen durch die Prozedur *match-type* beachtet werden. Durch die Reduzierung auf zulässige Intervallen in Verbindung mit der Eliminierung der überflüssigen Elemente in den Mengen werden die Anfragen auf eine minimale Form gebracht. Eine Anfrage wird nicht als Zeichenkette oder XPath-Baumrepräsentation aufgefasst, sondern durch Erfassung aller wirklich notwendigen Teilziele. Die Reihenfolge der enthaltenen Bedingungen, redundante und überflüssige Teilziele werden dadurch ignoriert. Dadurch wird erreicht, dass zwei Anfragen als identisch gelten, wenn ihre Ergebnismengen bezüglich der zu Grunde liegenden Semantik identisch sind.

Durch die Auflistung werden alle möglichen Situationen, die im *match-type*-Algorithmus nach Aufruf der Prozedur *build-ranges* für die *ranges* einer Anfrage vorliegen können, beschrieben. Auf eine algorithmische Beschreibung der modifizierten *match-type*-Prozedur wird hier ganz verzichtet. Der Ablauf bleibt der gleiche wie bei den Algorithmen in den Abbildungen 12 (Seite 48) und 14 (Seite 53).

Der Ablauf wird noch einmal kurz zusammengefasst: Die *ranges* der bearbeiteten Anfrage und die des untersuchten Cache-Eintrages werden gebildet und im Anschluss gegeneinander geprüft. Dabei werden alle möglichen Kombinationen von Situationen, definiert durch die soeben erfassten Regeln, zwischen den gebildeten *ranges* geprüft. Der Test beginnt wieder bei der speziellsten Situation (beide Anfragen schränken das betrachtete Attribut auf einen Wert ein, d.h. für beide gilt $min + 1 = max$) und endet mit Erfassen vorliegender Teilmengenbeziehungen zwischen den gebildeten Mengen $Sim, UnSim$ und $UnEq$. Die Überprüfung auf „exact match“ und „disjoint match“ erfolgt im Wesentlichen analog dem Test auf Erfüllbarkeit in der Prozedur *build-ranges*, kann also leicht anhand Tabelle 4 und den daraus folgenden Regeln nachvollzogen werden. Der Test auf „containing match“ und „contained match“ ist ebenfalls leicht nachvollziehbar: Neben dem Test, ob sich die gebildeten gültigen Intervalle $[min, max)$ gegenseitig enthalten und der Kontrolle der beiden $UnEq$-Mengen, müssen zusätzlich die Mengen Sim und $UnSim$ geprüft werden. Diese Mengen schränken das Anfrageergebnis ein, indem sie die gültigen Attributwerte auf eine bestimmte Menge von Worten eingrenzen, bzw. eine solche Menge ausgrenzen. Die Überprüfung resultiert dementsprechend in der Kontrolle von entsprechenden Teilmengenbeziehungen zwischen diesen Mengen der beiden gebildeten *ranges*.

Die resultierende Komplexität des Algorithmus wird wiederum nicht näher betrachtet. Der Unterschied zum Ablauf des allgemeineren Algorithmus ohne Unterstützung des Textähnlichkeitsoperators wird hier nur durch die neu zu verwaltenden Mengen bestimmt. Wie schon in Abschnitt 5.1 kurz erwähnt wurde, führen diese Mengen zu einer quadratischen asymptotischen oberen Schranke für die Komplexität. Der ’$\sim=$’-Operator bedeutet im Vergleich zum ’$\neq$’-Operator einen Mehraufwand, da die verwalteten Mengen Sim und $UnSim$ mehr Elemente enthalten können als die Menge $UnEq$. So schließen sich Teilbedingungen, die den ’$\sim=$’-Operator enthalten, in der Regel nicht aus. Für jedes Teilziel wird also ein neuer Wert in eine der Mengen aufgenommen, die in folgenden Schritten wiederum alle betrachtet werden müssen. Auch Operatoren wie ’$<$’ und ’$>$’ schränken die Mengen genrell nicht ein, wie bereits erläutert wurde. Bei der Implementierung sollten die genutzten Verfahren zur Realisierung der Mengen sorgfältig überdacht werden.

Die Festlegung auf XPaths *contains*-Funktion führt dazu, dass die tatsächliche Semantik des Textähnlichkeitsoperators nicht korrekt erfasst wird. Da er in den verschiedenen Quellsystemen auch auf anderen Beziehungen als auf enthaltenen Zeichenketten basieren kann, entsprechen während der Suche im Cache kombinierte Ergebnismengen nicht immer den vollständigen Ergebnismengen der Anfrage. Betrachtet man nur eine Quelle, die den Operator auf gleiche Weise umsetzt, sind die gesammelten Ergebnisse vollständig. Quellen, die den Operator nicht auf diese Weise realisieren, geben Ergebnisobjekte zurück, die mit dem beschriebenen Verfahren nicht als Teile des angefragten Ergebnisses erkannt werden. So kann es vorkommen, dass einige Teile eines im Cache gespeicherten Eintrags als Teilmenge korrekt erkannt werden, einige aber nicht, denn durch das Caching auf Ebene der Konzepte vereint ein Cache-Eintrag die Ergebnisse bezüglich mehrerer Quellsysteme, die das Konzept unterstützen. Andererseits kann auch die Situation entstehen, dass eine Ergebnismenge als komplett erkannt wird, diese aber nicht wirklich vollständig ist.

Das ist der Fall, wenn die Vereinigung mehrerer Cache-Einträge während der Suche als komplettes Ergebnis erkannt wird, die Ausführung der Anfrage auf den Quellsystemen aufgrund abweichender Semantik allerdings zusätzliche Ergebnisobjekte liefert. Da die Inklusion von Zeichenketten im YACOB-Mediator als eine mögliche Realisierung der unscharfen Suche unterstützt wird, sind die bestimmten Teile der Ergebnismengen aber immer wahre Teilmengen, die Bestimmung erfolgt korrekt. Die Folge allerdings ist, dass die gebildete Komplementäranfrage nicht mehr genutzt werden kann, um fehlende Daten von den Quellsystemen zu erhalten. Die gebildete Anfrage beschreibt die Beziehungen zwischen den Cache-Einträgen und der ausgeführten Anfrage, nimmt aber nicht Bezug auf die eventuell nicht erfassten Daten eines Eintrags. Die ihr entsprechende Ergebnismenge umfasst somit wiederum nicht alle fehlenden Daten.

Die einzige Lösung ist hier, immer die Originalanfrage anstatt der komplementären an die Quellsysteme zu leiten und die erhaltenen Daten erst nach Eliminierung aller schon als Teilergebnis vom Cache bestimmten Duplikate dem Anwender als fehlende Daten zurückzugeben. Vor der Speicherung im Cache muss gewährleistet sein, dass alle zur neuen semantischen Region gehörenden Daten vorliegen, aber keine Daten redundant im Cache gespeichert werden. Die im Cache vorliegenden Einträge enthalten dadurch auch wirklich die Daten, die ihren Beschreibungen, den ausgeführten Anfragen, bei Verwendung von CQuerys Textähnlichkeitsoperator entsprechen. Da bei der Suche im Cache und zum Selektieren von Daten aber XPaths *contains*-Funktion genutzt wird, könnte man sogar mit Ausführen der einem Eintrag zugeordneten Anfrage gegen die ihm zugeordneten Daten in der Regel nicht die komplette zugeordnete Ergebnismenge erfassen. Die Ausführung der Originalanfrage ist hier die einzige Möglichkeit, Ergebnisdaten, die einer anderen Semantik entsprechen, trotzdem den Cache-Einträgen zuordnen zu können. Das Vorgehen zur Duplikateliminierung und weitere Aspekte der Speicherung dieser neuen Daten werden in Abschnitt 5.5 behandelt.

Die Probleme bei der Realisierung des Verfahrens machen nun sogar das Ausführen der Originalanfrage im Fall eines „containing match“ nötig. In diesem Fall könnte die Suche im Cache eigentlich abgebrochen werden, da die dem betrachteten Cache-Eintrag zugeordneten Daten eine Obermenge der angefragten Ergebnismenge bilden. Da die Einträge auch wirklich alle Daten entsprechend CQuerys Ähnlichkeitsoperator enthalten, ist es sicher, dass alle Objekte der Ergebnismenge bereits im Cache gespeichert sind. Allerdings ist es nicht möglich, auch alle zugehörigen Daten zu selektieren, da nur XPath als Anfragesprache zur Verfügung steht. Alle Daten des Cache-Eintrages können allerdings auch nicht zur Ergebnismenge hinzugefügt werden, da dies einer falschen Antwort entsprechen würde. Auch in diesem Fall kann nur die Beantwortung der Originalanfrage durch die Quellen und anschließende Duplikateliminierung garantieren, dass alle fehlenden Daten erfasst wurden. Im Cache sind in diesem Fall allerdings keine neuen Daten zu speichern, denn diese wären absolut redundant. Der Cache-Eintrag wurde schon bei seiner Speicherung mit fehlenden Daten vervollständigt. Problematisch wird es, wird die dem Eintrag zugehörige Ergebnismenge in weiteren Schritten zusätzlich mit dem ’$\sim=$’-Operator eingeschränkt. Dann müssen die eigentlich überflüssigen Originalanfragen immer wieder neu ausgeführt werden. Hier könnte eine Verwendung von logischen Verweisen, wie sie auch in Abschnitt 4.3 bei der Skizzierung einer hierarchischen Regionen-Verwaltung angedeutet wurde, helfen, die Antwortzeit zu verringern.

Wichtig für angepassten Vorgehensweisen ist, dass Datenredundanz möglichst vermieden wird. In jedem Fall kann ein Teil der vorliegenden Daten ohne weitere Verzögerung vom Cache an den Anwender übergeben werden: Die Daten, die einer Ähnlichkeit auf Basis der beschriebenen Inklusion von Zeichenketten entsprechen, sind nach wie vor selektierbar.

Vermeiden lässt sich das Ausführen der Originalanfrage nur dann, wenn das komplette Ergebnis zu einer bearbeiteten Anfrage bei der Suche im Cache allein mit Hilfe von „contained match“- und/oder „exact match“-Einträgen gebildet werden kann. Da Cache-Einträge vor ihrer Speicherung im Cache immer durch Daten, die der CQuery-Semantik entsprechen, vervollständigt werden und bei diesen Beziehungen zwischen den Anfragen immer die gesamten Datenmengen der Einträge Teilmengen des Ergebnisses sind, wird die Beschränkung auf die Ausführung von XPath-Anfragen hier nicht zum Problem. Durch die ohne Selektion mögliche Kollektion der Ergebnisdaten gewinnt ein im Cache gefundenes „contained match“ hier sogar an Attraktivität. Zuvor wäre ein „containing match“ der optimalere Fall gewesen, durch die dann notwendige Ausführung der Originalanfrage entsteht aber ein zu vermeidender Mehraufwand.

Am Ablauf der Prozedur *cache-lookup*, wie er in Abschnitt 5.2 vorgestellt wurde, ändert sich prinzipiell nichts, die Unterstützung des Textähnlichkeitsoperators nimmt nur Einfluss auf die Bestimmung der Beziehung zwischen zwei Anfragen durch Aufruf der Prozedur *macth-type*. Nach wie vor werden die im Cache gespeicherten Einträge nacheinander gegen die einzelnen konjunktiven Ausdrücke der Anfrage geprüft und Teile der Ergebnismenge gesammelt. Der Cache kann also immer noch die Verringerung der Antwortzeit unterstützen, da Teile der Ergebnismnenge schnell (ohne nötige Quellanfrage) erkannt und dem Anwender geliefert werden können. Ob die gelieferten Daten komplett sind, und wenn nein, die Menge fehlender Daten, kann allerdings erst nach der Duplikateliminierung bestimmt werden, also erst nach Ausführung der Quellanfragen. Die Originalanfrage muss ausgeführt werden, sobald bei einem Suchlauf mindestens ein „overlapping match“ oder “containing match“ gefunden wurde, denn dann besteht die Gefahr, dass im Cache vorliegende Daten fälschlich nicht als Teil der Ergebnismenge erkannt wurden. Ob tatsächlich neue Daten im Cache zu integrieren sind, hängt davon ab, ob nach Beenden der Suche eine globale Komplementäranfrage resultiert. Die Notwendigkeit zur Bildung der Komplementäranfrage im Verlauf der Anfragebearbeitung in jedem Fall, wurde schon bei der Erläuterung der Verwaltung der semantischen Regionen in Abschnitt 4.3 erkannt. Eine resultierende Komplementäranfrage beschreibt die neu entstehenden semantischen Regionen und zeigt somit an, wenn tatsächlich Daten im Cache fehlen.

Das Ausführen der Originalanfrage anstatt der gebildeten komplementären Anfrage birgt auch einen kleinen Vorteil: das in Abschnitt 4.5 beschrieben Kohärenzproblem wird ohne zusätzliche Arbeit teilweise gelöst. Da immer die aktuellen Daten bei den Quellen angefragt werden, kann es zumindest in diesem Fall nicht passieren, dass der Anwender auf seine Anfrage unvollständige Daten erhält (bei identischen Anfragen besteht das Problem nach wie vor). Durch das Verfahren werden Daten, die in den Quellen neu hinzugefügt worden sind, im Cache aber eventuell falschen Regionen zugeordnet.

Diese neuen Daten verbleiben nach der Duplikateliminierung in der Ergebnismenge der Komplementäranfrage, die eine neue semantische Region beschreibt. Damit gewinnt die Wahrung der Aktualität der Cache-Einträge für diesen Fall zunehmend an Bedeutung. Wie die Ausführung der Originalanfrage mit einem guten Kohärenzprotokoll zu verknüpfen ist, könnte Inhalt zukünftiger Arbeiten sein.

5.5 Speicherung komplementärer Daten

In diesem Abschnitt wird abschließend kurz beschrieben, wie neue Daten im Cache gespeichert werden. Diese Daten entsprechen immer der zum Cache-Inhalt komplementären Ergebnismenge bezüglich einer ausgeführten Anfrage q und werden im Folgenden einfach als *komplementäre Daten* bezeichnet. Jeder konjunktive Teilausdruck der Komplementäranfrage beschreibt eine neue semantische Region. Im einfachsten Fall, dass kein mit der Anfrage in Zusammenhang stehender Cache-Eintrag gefunden wurde, entspricht die beschreibende Komplementäranfrage der originalen. In diesem Fall entsprechen die zu speichernden Daten komplett der ausgeführten Anfrage und der (den) von ihr beschriebenen semantischen Region(en). Besondere Anforderungen bei der Speicherung der Daten entstehen durch die in Abschnitt 5.4 beschriebene Unterstützung des Textähnlichkeitsoperators. Wesentliche Aufgaben sind, garantieren zu können, dass die Daten eines Cache-Eintrages auch wirklich bezüglich der CQuery-Semantik vollständig sind und dass Datenredundanz trotzdem weitgehenst vermieden wird.

Als Erstes kann die Duplikateliminierung vorgenommen werden. Dieser Schritt gehört nicht nur zur Speicherung komplementärer Daten, denn durch die Probleme, die aus der Unterstützung des Textähnlichkeitsoperators rühren, müssen Duplikate gegebenenfalls auch eliminiert werden, wenn keine neuen Daten im Cache zu speichern sind. Die erhaltene Datenmenge wird gegen die schon dem Anwender übermittelte Teilmenge geprüft und Duplikate werden entfernt. Die Art und Weise des anzuwendenden Vorgehens ist Entscheidung der Implementierung. Zu bedenken ist, dass die Duplikateliminierung für XML-Daten keinesfalls trivial ist. Nach erfolgter Eliminierung aller Duplikate kann die verbleibende Ergebnismenge dem Nutzer des Systems übergeben werden. Die noch ausstehende Zuteilung der Daten zu neuen Cache-Einträgen erfolgt somit ohne weitere Verzögerung der Antwortzeit. Noch sind allerdings nicht alle Daten aus der Ergebnismenge entfernt worden, die schon im Cache vorliegen. Wurden während der Suche im Cache zur Bildung des zurückgegebenen Teilergebnisses auch Einträge genutzt, auf deren Datenmenge dafür XPath-Anfragen ausgeführt wurden („containing match“ und „overlapping match“), so wurden Teile dieser Daten eventuell fälschlich nicht zur Ergebnismenge zugeordnet, obwohl sie ein Teil davon sind. In diesem Fall sind sie in der nach der ersten Duplikateliminierung gewonnenen Ergebnismenge enthalten und würden bei Eintrag im Cache zu Datenredundanz führen. Dieser Umstand macht einen zweiten Schritt zur Duplikateliminierung nötig, der erst später ausgeführt wird und dessen Erläuterung noch folgt. Wurde die Originalanfrage hingegen nur ausgeführt, um die Vollständigkeit einer generierten Ergebnsimenge zu prüfen, so müssen die nächsten Schritte nicht ausgeführt werden, denn keine neuen Daten sind im Cache zu speichern.

Im nächsten Schritt muss die disjunktive Normalform (DNF) der Komplementäranfrage gebildet werden, denn im Cache werden nur konjunktiven Ausdrücken entsprechende Einträge gespeichert. Nach Bildung der DNF müssen die Daten dann den resultierenden Teilausdrücken zugeordnet werden. Dabei wird aber sofort wieder die Unterstützung des Textähnlichkeitsoperators zum Problem. Da nicht alle einer Teilanfrage zugehörigen Daten im Cache korrekt erkannt werden können (siehe dazu Abschnitt 5.4), ist es nicht möglich, die Daten entsprechend den Teilausdrücken zu teilen. Zudem würden die Teile der gebildeten Anfrage nicht genau den Teilen der zuvor gebildeten Komplementäranfrage entsprechen, denn diese weist gegenüber ihrer DNF eine für die weitere Behandlung optimierte Form auf (siehe Abschnitt 5.3). Die Problematik wird hier zunächst sehr einfach gelöst: Sämtliche Anfragen werden zur Beantwortung in konjunktive Teilausdrücke zerlegt und diese einzeln zur Beantwortung an die Quellen gesandt. Diese Teilausdrücke entsprechen entweder den Teilen der DNF, wenn keine Komplementäranfrage gebildet wurde weil kein überlappender Eintrag gefunden wurde, oder den Teilen der zuvor gebildeten Komplementäranfrage. Dadurch entsteht ein zusätzlicher Aufwand, erhaltene Daten müssen zur Speicherung im Cache aber nicht in Mengen geteilt werden, die den konjunktiven Ausdrücken entsprechen. Dieser Schritt entfällt damit also vorerst, der Bedarf nach besseren Lösungen, um den Mehraufwand in der Anfragebearbeitung zu vermeiden, allerdings besteht. So ist es nicht zwingend erforderlich, die Aufteilung durch getrennte Ausführung der Teilanfragen zu erzielen. Ebenso kann diese Aufgabe nach Beantwortung der gesamten Quellanfrage auch vom Mediator übernommen werden.

Wurde bei dem zugehörigen Suchlauf im Cache kein überlappender Cache-Eintrag gefunden, entspricht die Komplementäranfrage also der originalen Anfrage und sind die Daten komplett im Cache in neuen Einträgen zu erfassen, werden die erhaltenen Daten einfach semantischen Regionen (und damit Cache-Einträgen) zugeordnet, die den Teilausdrücken der Komplementäranfrage entsprechen. In diesem Fall ist die Speicherung der komplementären Daten abgeschlossen, die neuen Einträge und ihre Beschreibungen können in erneute Suchen integriert werden. Wurde allerdings eine Komplementäranfrage gebildet, so müssen die zur Originalanfrage erhaltenen Daten vor Speicherung im Cache noch entsprechend den Teilausdrücken dieser komplementären Anfrage bearbeitet werden. Ob dem so ist, muss bis zum Abschluss der Speicherung der fehlenden Daten im Cache temporär festgehalten werden. Zusätzlich wird eine Zusammenfassung der Teile der gebildeten Komplementäranfrage mit Zuordnung zur beantworteten Originalanfrage benötigt. Wie das realisiert wird ist wiederum Sache der Implementierung.

In Abbildung 18 auf Seite 65 wurde bereits die schrittweise Bildung der Komplementäranfrage illustriert. Jede der am Ende resultierenden Teile beschreibt eine neue semantische Regionen. Welche semantischen Regionen (Cache-Einträge) zur Bildung dieser finalen Regionen kombiniert (und die zugehörigen Anfragen negiert) wurden, wird für jeden einzelnen Schritt ebenfalls im Cache zwischengespeichert. Anhand dieser *Cache-Geschichte* kann nun auch der zweite Schritt der Duplikateliminierung erfolgen. Alle Cache-Einträge, die zuvor zur Kombination der Teilergebnismenge genutzt worden, indem einige ihrer Daten mit XPath-Anfragen selektiert worden, müssen dazu betrachtet werden. Das sind wiederum alle bei der Suche gefundenen Einträge, für die ein „containing match“ oder „overlapping match“ als Beziehung zur Anfrage festgestellt wurde.

Jede Datenmenge der besuchten Cache-Einträge wird gegen die aktuelle Ergebnismenge geprüft und gefundene Duplikate werden wiederum eliminiert. Dieser Schritt ist absolut notwendig, um Datenredundanz im Cache zu vermeiden. Ob es unter Umständen doch sinnvoller ist, diesen zweiten Schritt der Duplikateliminierung mit dem ersten zu vereinen, können nur entsprechende Experimente zeigen. Da das Erzielen einer minimalen Antwortzeit allerdings Priorität ist, müsste der benötigte Aufwand schon erheblich sinken, um eine solche Maßnahme zu rechtfertigen.

Der letzte noch auszuführende Schritt ist die Zuordnung der verbleibenden Daten zu den Teilausdrücken der Komplementäranfrage. Bei der Zuordnung der Daten entstehen durch den '$\sim=$'-Operator allerdings Probleme, genau wie bei Zuteilung der Daten zu den Teilen der DNF, die deshalb vermieden wurde. Nicht alle Daten können den entsprechenden semantischen Regionen mit XPath-Anfragen zugeordnet werden. Die bis zu diesem Punkt nicht eliminierten Daten entsprechen der Ergebnismenge eines einzelnen Teilausdruckes der Originalanfrage. Jeder dieser Teilausdrücke hat im Cache potentiell mehrere neue semantische Regionen „erzeugt“. Diese Regionen, bzw. ihre Beschreibungen (die Anfragen), weisen eine gewisse „Verwandtschaft“ auf. Dies beruht auf der in Abschnitt 5.3 ausführlich beschriebenen Bildung der Komplementäranfrage, denn alle zu einem Teilausdruck der Originalanfrage gebildeten Teile der komplementären Anfrage weisen einen gleichen „Stamm“ auf, d.h. sie alle enthalten eine bestimmte Menge von Teilzielen gemeinsam. Der Unterschied drückt sich darin aus, dass für in einem Schritt nacheinander gebildete Teile gilt: enthält Teil i das Teilziel j des überlappenden Cache-Eintrages in negierter Form, so enthalten alle Teile $k : k > i$ das Teilziel j in Originalform. Die Folge ist, dass die zuerst gebildeten Teile allgemeiner als die letzteren sind und größere semantische Regionen beschreiben. Größer soll hier anzeigen, dass die Regionen durch weniger Teilziele (also in weniger Dimensionen) eingeschränkt sind und nicht Bezug auf die tatsächliche Anzahl enthaltener Datenobjekte nehmen.

Der Zusammenhang unter solchen Regionen kann bei der Zuordnung der Daten ausgenutzt werden. Die Daten können schrittweise, entsprechend den gespeicherten Schritten bei Bildung der Komplementäranfrage, den resultierenden Anfrageausdrücken zugeteilt werde. Dazu wird jeder gebildete Teil als XPath-Anfrage an die Datenmenge gerichtet und das darauf erhaltene Ergebnis der jeweiligen Region zugeordnet. Der letzten gebildeten semantische Region eines jeden Schrittes allerdings werden die verbleibenden Daten zugeordnet, nicht die ihrer XPath-Anfrage entsprechenden. Daten, die durch die Semantik der XPath-Anfragen fälschlich nicht bereits anderen Regionen zugeordnet wurden, „landen“ immer in der letzten der gebildeten, der speziellsten und kleinsten semantischen Region. Den vorher gebildeten allgemeineren Teilen werden Objekte allerdings auch fälschlich zugeordnet. Dies kann leider auch zur fehlerhaften Beantwortung von Anfragen führen. Ein recht einfaches kleines Beispiel zur Veranschaulichung:

Die bearbeitete Anfrage q sei:

```
//Grafik[Kuenstler~='Gogh']
```

Der gerade betrachtete überlappende Cache-Eintrag c sei:

```
//Grafik[Titel~='Personen' and Titel~='Portrait']
```

Bei Bildung der Komplementäranfrage entstehen zwei Teilausdrücke. Der erste Ausdruck $\overline{q_1}$ ist:

```
//Grafik[Kuenstler∼='Gogh' and Titel≁='Personen']
```

Als zweiter Teilausdruck $\overline{q_2}$ entsteht:

```
//Grafik[Kuenstler∼='Gogh' and Titel≁='Portrait' and
                Titel∼='Personen']
```

Ein Objekt X, das nun im Attribut Titel einen Wert aufweist, der fälschlich als nicht ähnlich zu 'Personen' erkannt wird (z.B. könnte das 'Person' sein), wird nun trotzdem der ersten gebildeten Region zugeordnet. Auf die neue Anfrage

```
//Grafik[Titel∼='Personen']
```

würde nun ein falsches Ergebnis im Cache bestimmt werden, denn nur $\overline{q_2}$ wird (und das ist korrekt!) als überlappender Eintrag erkannt. Allerdings wird das Objekt X nun nicht mit erfasst, da es fälschlich $\overline{q_1}$ zugeordnet wurde und dieser Teil hier (ebenfalls korrekt) als „disjoint match" erkannt wird. Ein solches Problem wird mit Ausführung der Originalanfrage und folgender Duplikateliminierung zumindest teilweise gelöst, denn dann wird X nach dem ersten Schritt der Duplikateliminierung als Teil der fehlenden Daten an den Anwender übergeben. Im zweiten Schritt jedoch wird es aus der Menge entfernt und somit nicht $\overline{q_2}$ zugeordnet. Das bestehende Problem ist die falsche Zuordnung von X zu $\overline{q_1}$.

Problematischer ist die Übermittlung von schlichtweg falschen Ergebnissen. Dies basiert aber in der Regel auf Anfragen ähnlicher Form wie

```
//Grafik[Titel≁='Personen'],
```

die in dieser Form selten zu erwarten sind, denn die Anfrageformulierung in der Benutzerschnittstelle des YACOB-Systems bietet die Möglichkeit nach der Suche über Ähnlichkeiten, nicht aber über Nicht-Ähnlichkeiten.

Hier könnte die beschriebene „Verwandtschaft" unter den zu einem Teilausdruck der Originalanfrage gebildeten komplementären Teilen von Vorteil sein, denn dadurch ist die resultierende Abweichung enthaltener Daten von der Regionen-Beschreibung eventuell geringer. Die Zuteilung der Daten zu den neuen semantischen Regionen könnte natürlich auch nur für die finalen, in den Cache zu integrierenden Regionen erfolgen. Diese Regionen werden durch die Teile der zuvor gebildeten globalen Komplementäranfrage beschrieben, also den Teilausdrücken der letzten Stufe der entsprechenden Baumgrafik. Ein Teil einer in einem frühen Schritt der Suche gebildeten Komplementäranfrage kann am Ende mehrere Teile der finalen Komplementäranfrage erzeugen (Abbildung 18 auf Seite 65 verdeutlicht das am besten). Werden die Daten also nur einmal entsprechend der finalen Regionen geteilt, so ist diese Aufteilung, aufgrund der höheren Anzahl in den Teilanfragen enthaltener Teilziele auf der letzten Stufe des entstehenden Baumes, eventuell wesentlich fehleranfälliger. Bei schrittweiser Zuteilung jedoch könnte es sein, dass fehlerhafte Daten besser verteilt werden. Diese Überlegung beruht auf zwei Faktoren.

Zum Einen ist die „Verwandtschaft“ unter den in einem Schritt gebildeten Teilausdrücken noch spezieller als unter denen der globalen Komplementäranfrage, zum Anderen enthalten diese Ausdrücke weniger Teilziele. Dies kann unter Umständen dazu führen, dass die Abweichungen von den XPath-Anfragen geringer sind und die Fehler besser über die verschiedenen Hierarchie-Stufen „verteilt“ werden. Ob die schrittweise Aufteilung analog zur Bildung der Komplementäranfrage die Fehlerquote bei der Zuteilung wirklich verringern kann oder ob sie nur unnötigen Mehraufwand darstellt, muss genauer untersucht und durch entsprechende Experimente bewertet werden.

Zusammenfassend kann festgehalten werden, dass die Unterstützung des Textähnlichkeitsoperators in der jetzigen Form zu erheblichem Mehraufwand und eventuell fehlerhaft gespeicherten Daten führen kann. Die Entstehung von Datenredundanz wird mit den beschriebenen Maßnahmen weitestgehend minimiert. Ob die Beschleunigung der Anfragebearbeitung eventuell auftretende Fehler und den Mehraufwand trotzdem überwiegt, können nur entsprechende Experimente zeigen.

6 Implementierung und Tests

Den abschließenden Teil der Arbeit bildet die Implementierung des entwickelten Caches. Dies ist zum Einen nötig, um eine Evaluation des Caches zu ermöglichen, zum Anderen soll der semantische Cache in das bereits entwickelte YACOB-Mediatorsystem integriert werden.

Dieses Kapitel soll die Umsetzung keinesfalls im Detail beschreiben. Es wird vielmehr ein Überblick über die bedeutendsten Aspekte und die allgemeine Struktur der Implementierung gegeben. Eine detaillierte Beschreibung würde hier den Rahmen überschreiten. Zudem werden die vorgestellten Prinzipien und Algorithmen größtenteils ohne jede Modifikation mit Hilfe zur Verfügung stehender Java-APIs umgesetzt. Wie das schon vorliegende Mediatorsystem wurde das vorgestellte Konzept des semantischen Caches komplett in Java implementiert. Für interessierte Leser befindet sich im Anhang dieser Arbeit eine CD, die die erstellten Klassen als Quellcode und als übersetzte Klassen (SDK 1.4.2) enthält. Auf dieser CD befindet sich ebenfalls die Klassendokumentation, sowie eine kurze Zusammenfassung zu Benutzung und Installation der Implementierung.

6.1 Design-Entscheidungen

Das beschriebene allgemeine Vorgehen beim semantischen Caching wurde im Verlauf dieser Arbeit schrittweise auf die spezielle Anwendung im YACOB-Mediatorsystem eingegrenzt. Die behandelten Einschränkungen und speziellen Voraussetzungen führten zu einigen Modifikationen und Erleichterungen am allgemeinen Konzept, die in den vorangegangenen Kapiteln ausführlich behandelt wurden. Einige Fragen und Entscheidungen sind allerdings offen geblieben, da sie für die vorgestellten Methoden und Algorithmen nicht entscheidend sind. Bei der Implementierung sind diese Details der Umsetzung nun genauer zu betrachten.

In diesem Abschnitt werden wiederum nur die für die Implementierung bedeutendsten Design-Entscheidungen erläutert, da eine Beschreibung aller zu betrachtenden Aspekte den Rahmen dieser Arbeit überschreiten würde.

Modifikationen am *match-type*-Algorithmus

Bei der Beschreibung der Anfragebearbeitung in Kapitel 5 wurde ein Algorithmus für das *Query Containment* vorgestellt, der die Beziehung zweier Anfragen mittels eines Laufs über alle bekannten Attribute bestimmt. Dies setzt voraus, dass die Menge aller einschränkbaren Attribute zu jedem Konzept im Voraus bekannt ist.

Durch das definierte Konzeptmodell ist die Bestimmung der jeweiligen Eigenschaften auch unproblematisch. Dennoch weist der implementierte Algorithmus aus Gründen der Effizienz ein leicht modifiziertes Vorgehen auf: Alle in der Anfrage enthaltenen Attributbelegungen werden nach dem Namen des Attributes in einer Hash-Tabelle gesammelt.

Stellt eine Anfrage a eine Bedingung an ein Attribut A, welches von einer weiteren Anfrage b nicht eingeschränkt wird, so hat b im Attribut A einen „allgemeineren Charakter“. Beim Vergleich der Anfragen wird dies durch das Fehlen einer Belegung für A in der zu b gebildeten Hash-Tabelle deutlich. Zuvor wurde dies durch initialisieren aller Attribute mit $(-\infty, \infty)$ bzw. (“”, $\emptyset$) (siehe Abschnitte 3.4 bzw. 5.1) erreicht.

Bei dem modifizierten Vorgehen Vorgehen wird eine Bestimmung der Menge aller zu einem bestimmten Konzept einschränkbaren Attribute überflüssig. Sollte sich diese einmal ändern, so muss die Implementierung nicht angepasst werden. Zudem ist zu erwarten, dass in vielen Anfragen von dieser Menge nur verhältnismäßig wenige Attribute gleichzeitig eingeschränkt werden. Eine Neubestimmung für jede eintreffende Anfrage stellt dann einen leichten Mehraufwand dar und wird vermieden.

Desweiteren wird die Beziehung zwischen Anfrage und Cache-Eintrag nicht wie geschildert in einer einzigen Prozedur bestimmt. Im Zuge der objektorientierten Umsetzung wurde das Vorgehen in Teilaufgaben unterteilt, die von Methoden verschiedener implementierter Datenstrukturen umgesetzt werden. Das Vorgehen ändert sich dadurch leicht im Ablauf, nicht aber das Prinzip, auf dem es beruht. Näheres wird hier nicht geschildert, genauere Informationen sind anhand der Klassendokumentation, die auf der CD im Anhang enthalten ist, ersichtlich.

Physischer Speicher

Um die Anforderungen an den physischen Speicher zu erfüllen, wird die XML-Datenbank XINDICE (ausgesprochen: „<zeen-dee-chay> in your best faux Italian accent“ - Originaltext der Homepage) verwendet. Diese Datenbank ist ein frei erhältlicher Vertreter der nativen XML-Datenbanken. Neben klaren Vorteilen wie Ausfallsicherheit und Skalierbarkeit, ist vor allem die integrierte XPath-Anfragekomponente entscheidend für die Wahl. Diese Form der Anfrage wird genutzt, wenn dem Ergebnis einer bearbeiteten Anfrage nur ein Teil der einem Cache-Eintrag zugeordneten Daten entspricht. Die entsprechenden Daten können dann einfach extrahiert werden, indem man die aktuelle Anfrage an den im Cache gespeicherten Eintrag richtet. Der Vorzug einer reinen XML-DB, gegenüber z.B. einer relationalen, ist im vorliegenden Anwendungsfall offensichtlich: die Daten liegen in XML vor, werden wieder in XML benötigt und zudem mittels XPath angefragt.

In XINDICE werden XML-Dokumente in sogenannten *Collections* gespeichert. Diese sind hierarchisch angeordnet, die Gliederung ist mit einem Verzeichnisbaum vergleichbar. Da das Caching auf Konzeptebene erfolgt, wird zu jedem angefragten Konzept eine solche Collection angelegt, die wiederum zwei *Sub-Collections* mit den Namen „entries“ und „results“ enthält. In „results“ liegen die bezüglich einer Anfrage gespeicherten Ergebnis-Dokumente. Die Beschreibung dieser Daten, konkret der Anfragestring zerlegt in die einzelnen Teilziele, die gebildeten *ranges* (siehe Abschnitte 3.4 und 5.1) sowie ein Verweis zum Ergebnisdokument) sind, ebenfalls XML-codiert, in „entries“ gespeichert. Wird nur ein Teil eines Dokumentes zur Zusammenstellung der Ergebnisdaten benötigt, so kann der entsprechende Teil effizient mittels einer XPath-Anfrage auf das verwiesene Dokument extrahiert werden.

Die Struktur der Datenbank ist zusammen mit der Anbindung des Caches an das Konzeptmodell in Abbildung 11 auf Seite 46 dargestellt.

Die Speicherung der einen Eintrag beschreibenden Daten zusammen mit den Ergebnisdaten in einer Datenbank ist naheliegend, da somit ein einheitlicher physischer Speicher genutzt werden kann. Die dadurch entstehende Anforderung ist, die beschreibenden Daten ebenfalls im XML-Format zu formulieren.

Um einen Cache-Eintrag in das XML-Format umwandeln und in der Datenbank abspeichern zu können, wird der java.beans.XMLEncoder verwendet, der eine einfache Serialisierung von Java-Klassen nach XML ermöglicht. Die Anbindung zur Datenbank wird im Programm mittels des XML:DB API umgesetzt, welches als standardisierte Schnittstelle für verschiedene XML-Datenbanken entwickelt wurde. Eine Vielzahl weiterer Informationen zur XINDICE-Datenbank und zum XML:DB-Projekt findet sich im Internet unter

`http://xml.apache.org/xindice/`
und
`http://www.xmldb.org/`.

Weitere besondere Aspekte

Verarbeiten von XPath-Anfragen Zum Verarbeiten der XPath-Anfragen können existierende *Parser* verwendet werden. Ein XPath-Parser liest eine Anfrage und bildet sie zur Verarbeitung auf eine geeignete Datenstruktur im Hauptspeicher ab. Allerdings wird CQuerys Textähnlichkeitsoperator von XPath nicht unterstützt. Zur Bearbeitung der XPath-Anfragen müssen in den Anfragen enthaltene '$\sim=$'-Operatoren in Funktionsaufrufe umgewandelt werden. Der Zugriff auf den verwendeten XPath-Parser und die Methoden zum Lesen und Verändern von Anfragen, werden in einer eigenen Klasse vereint.

Verwaltung einer *Cache-Geschichte* Um als Antwort auf Originalanfragen erhaltene komplementäre Daten im Cache speichern zu können, ist die Verwaltung einer *Cache-Geschichte* nötig. Realisiert wird dies über IDs, die von der Cache-Verwaltung mit eventuellen Ergebnisdaten und/oder komplementären Anfragen an den Aufrufer zurückgegeben werden. Diese IDs dienen später dazu, die zu einer ausgeführten Suche gehörigen Komplementäranfragen, überlappenden Cache-Einträge und gesammelten Daten zuordnen zu können.

Duplikateliminierung Die Duplikateliminierung wurde nicht implementiert. Es ist auch nicht vorgesehen, sie innerhalb des Caches auszuführen. Diese Aufgabe wird vom Mediator übernommen und durch entsprechende Methodenaufrufe erfüllt. Da der Cache noch nicht vollständig im Mediatorsystem eingebunden ist, wurde auf die Eliminierung aller Duplikat bei Durchführung der Experimente in den folgenden Abschnitten in der Regel verzichtet. Nur in notwendigen Fällen wurde sie durch ein aufwendiges manuelles Verfahren ersetzt. Dies ist z.B. zur Bestimmung der exakten Cache-Größe erforderlich.

Mengen-Implementierung Die Realisierung der in den Abschnitten 5.1 und 5.4 eingeführten Mengen zur Lösung der Frage des *Query Containment* erfolgt zunächst mit Hilfe des objektorientierten Datentyps *LinkedHashSet.* Das ist eine von Java zur Verfügung gestellte Klasse zur assoziativen Speicherung von Objekten, in der die Objekte zusätzlich durch eine verkettete Liste verbunden sind. Durch die Kombination dieser Eigenschaften wird versucht, den bei der Anfragebearbeitung resultierenden Aufwand zu minimieren. Die Vorteile anderer effizienter Möglichkeiten sind zu untersuchen, wenn der Cache nach endgültiger Einbindung in das YACOB-Mediatorsystem optimiert wird.

6.2 Die Klassenhierarchie im Überblick

Die implementierte Klassenhierarchie ist leicht durch ein UML-Diagramm darstellbar. Die Abbildung des kompletten Diagramms wäre allerdings sehr komplex und zu detailliert. Der hier in Abbildung 19 dargestellte Teil des Klassendiagramms stellt nur die wichtigsten Klassen mit den wesentlichen Methoden und Attributen dar. Die Abbildung soll die generelle Struktur sowie die Beziehungen und Art und Weise der Kommunikation unter den Klassen illustrieren. Sie ist aber nur als Skizze der implementierten Hierarchie zu verstehen, da detaillierte Informationen fehlen.

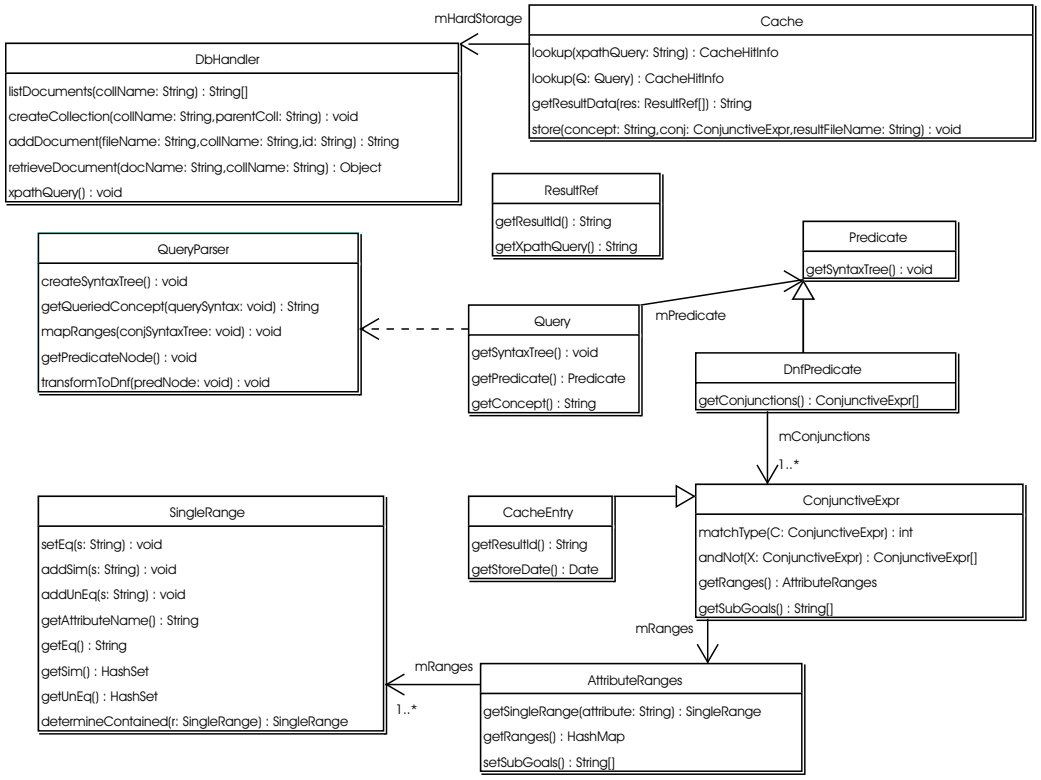


Abbildung 19: Ausschnitt des UML-Diagramms der Implementierung

Den eigentlichen Cache repräsentiert die Klasse *Cache*. Sie bietet die nötigen Methoden zur Suche und Speicherung neuer Daten im Cache an, aber auch zur Evaluation benötigte Statistikmethoden. Die Klasse *Query* stellt die zur Suche übergebenen Anfragen dar. Eine Anfrage enthält ein Prädikat (*Predicate*), dies ist eventuell in disjunktiver Normalform (*DnfPredicate*). Die Bestimmung der Beziehung zwischen zwei Anfragen erfolgt auf den einzelnen im Prädikat enthaltenen konjunktiven Teilausdrücken (*ConjunctiveExpr*). Ein Eintrag im Cache (*CacheEntry*) ist ein spezieller konjunktiver Ausdruck, der zusätzlich Verwaltungsinformationen, wie z.B. einen Ergebnisverweis enthält. Die eigentliche Bestimmung des *match-type* wird an die Klassen delegiert, welche die *ranges* repräsentieren: *AttributeRanges* und *SingleRange*. Die Klassen *DbHandler* und *QueryParser* stellen Methoden für den Datenbankzugriff und zum Lesen und Verarbeiten von Anfragen bereit. Der Vorteil dieser Ausgliederung ist, dass der verwendete XPath-Parser und die zu Grunde liegende XML-Datenbank leicht ausgetauscht werden können.

6.3 Experimente und Simulation

In diesem Abschnitt werden abschließend die Ergebnisse einiger erster Simulationen präsentiert. Diese ersten Ergebnisse sollen es ermöglichen, den realisierten Cache zu evaluieren. Dabei soll nicht nur der Nutzen des semantischen Cachings im Allgemeinen bewertet werden, sondern auch die in diesem Beitrag erarbeitete Realisierung im Speziellen. Die ausgeführten Experimente können allerdings nur einen ersten Überblick über das Leistungsverhalten des Caches geben. Detaillierte Simulationen in realen Umgebungen müssen die hier erzielten Resultate noch bestätigen. Da zur Zeit aber zum Einen keine Informationen über das typische Verhalten der Anwender des YACOB-Mediatorsystems verfügbar sind und zum Anderen der Cache noch nicht vollständig in das System integriert ist, können nur erste vorläufige Tests zur Evaluation herangezogen werden. Die erzielten Resultate werden grafisch dargestellt, eine genaue Auflistung der gemessenen Werte erfolgt nicht.

Simulationsumgebung und Durchführung

Die Experimente wurden auf einem Rechner mit schnellem Zugang zum Universitäts-Netz ausgeführt. Eine gute Netzanbindung war somit gegeben. Die beschränkte Hardware des genutzten Rechners allerdings führte in einigen Fällen zu Verzögerungen im Ablauf, vor allem bedingt durch den mit 256 MB begrenzten Hauptspeicher. Da die ganze Implementierung, ebenso wie z.B. der genutzte Web-Server, auf stellenweise recht speicherintensiven Java-Anwendungen basiert, wurden die Leistungsgrenzen des genutzten Rechners oft erreicht.

Der Cache lief in den Versuchen unabhängig vom YACOB-Mediatorsystem. Zur Beantwortung der Testanfragen wurden die Wrapper von drei unterschiedlichen Quellsystemen genutzt, die auch schon in Abschnitt 2.2 erwähnt worden: `www.lostart.de`, `www.herkomstgezocht.nl` und `www.restitution-art.cz`. Leider ließen sich nicht alle drei Quellsysteme in der gleichen Art und Weise für die Tests nutzen.

Durch beschränkte Anfragemöglichkeiten und der begrenzten Verfügbarkeit von Daten waren einige Tests nur in beschränktem Maße möglich. Gerade die Lostart-Quelle erwies sich hier als die stabilste der genutzen Quellen, allerdings auch als die schnellste. Die Wrapper der Quellen wurden einzeln von der Cache-Verwaltung angesprochen. Konnte eine Anfrage von einer Quelle nicht beantwortet werden, so wurde die entsprechende Antwortzeit anhand bereits gemessener Werte und dem Verhalten der anderen Wrapper approximiert. Die einzelne Nutzung der Wrapper entpricht eher dem Caching auf Quellenebene als auf Konzeptebene. Durch die gegebenen technischen Voraussetzungen war dies aber nicht anders zu lösen. Im Cache wurden eventuelle Ergebnisdaten, wie zuvor beschrieben, getrennt für die einzelnen Konzepte verwaltet.

Auf eine Duplikateliminierung zwischen der im Cache bestimmten Ergebnismenge und der auf die Quellanfragen erhaltenen Antwort wurde verzichtet. Die erzielten Zeiten sind somit als leicht optimiert anzusehen, denn der Aufwand für die Duplikateliminierung kann die Antwortzeit durchaus beeinflussen. Zusätzlich positiv wirkt sich für die gemessenen Zeiten aus, dass der Cache-Rechner zugleich der Versuchs-Rechner ist. Nach Eintreffen der vollständigen Antwort im Cache ist bis zum Erhalten der Daten somit kein Netzwerkverkehr mehr nötig gewesen. Die Speicherung der Daten im Cache erfolgt ebenfalls nach Übermittlung des Ergebnisses, d.h. nach der Zeitnahme, so wie es auch für den Cache im YACOB-Mediator geplant ist.

Anfragemix

Wie bereits erwähnt wurde, können die hier erfassten Experimente nur als vorläufige Tests angesehen werden. Um einen wirklich aussagekräftigen Anfragemix für die Durchführung einer Simulation zu erhalten, ist zuvor die Analyse des Nutzerverhaltens über einen längeren Zeitraum nötig.

Hier wurde zunächst ein einfacher Mix aus 13 Anfragen gewählt. Diese Anfragen wurden aus kleinen Protokollen zu Testsitzungen im YACOB-Mediator gewonnen. Auf eine vollständige Auflistung wird verzichtet, es folgen beispielhaft einige der Anfragen:

```
//Malerei[titel~='land' and person~='Malchin']
            //Malerei[person~='Gogh']
 //Zeichnung[(person~='Dürer' or person~='Bol') and
                titel~='Portrait']
//Zeichnung[titel~='Portrait' and titel~='Woman' and
                person~='August']
                        ⋮
```

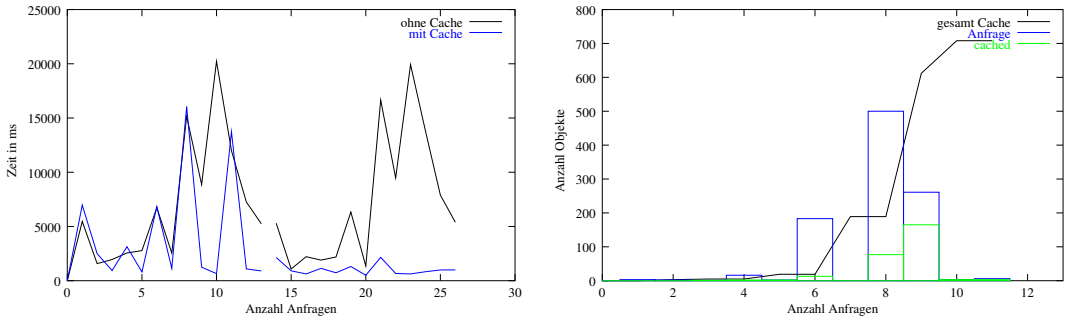
In dem Anfragemix sind ausschließlich Anfragen enthalten, die den Textähnlichkeitsoperator '~=' nutzen. Anfragen mit '=' sind äußerst ungeeignet für erste Tests, weil die darauf erhaltenen Ergebnismengen in den meisten Fällen leer sind. Nähere Ausführungen dazu sind in den vorangegangenen Kapiteln zu finden.

Alle Versuche wurden mit einem zunächst leeren Cache durchgeführt, der sich erst im Lauf der Zeit füllen konnte. Zusätzlich zu den Versuchen mit dem vorgestellten Anfragemix, wurden noch weitere Tests mit anderen Anfragen realisiert.

Für diese Tests wurde ein jeweils einschränkender und ein erweiternder Anfragemix generiert. Jede der zuvor formulierten 13 Anfragen wurde dafür einzeln durch Teilbedingungen eingeschränkt bzw. erweitert (wobei nicht auf Einschränkung oder Erweiterung bezüglich der Anfragen untereinander geachtet wurde). Der resultierende Mix wurde dann mit Cache-Unterstützung ausgeführt. Der Inhalt des Caches entsprach vor Ausführung der ersten Anfrage dabei immer dem Zustand, in den er durch Bearbeitung des ersten Anfragemixes gebracht wurde. Dadurch soll das erwartete Nutzerverhalten simuliert werden.

Um die erzielten Resultate zu bestätigten, wurde auch versucht, einen größeren Anfragemix zu erstellen um das Verhalten des Caches im Zeitablauf besser analysieren zu können. Die zuvor verwendeten Anfragen wurden kombiniert und durch weitere Bedingungen zufällig modifiziert. Leider traten hier die erwähnten Probleme mit den Quellen auf, welche die Anfragen stellenweise gar nicht beantworten konnten. Der gebildete Anfragemix konnte nur auf der Lostart-Quelle vollständig ausgeführt werden, lieferte aber auch hier viele leere Ergebnisse.

Ergebnisse

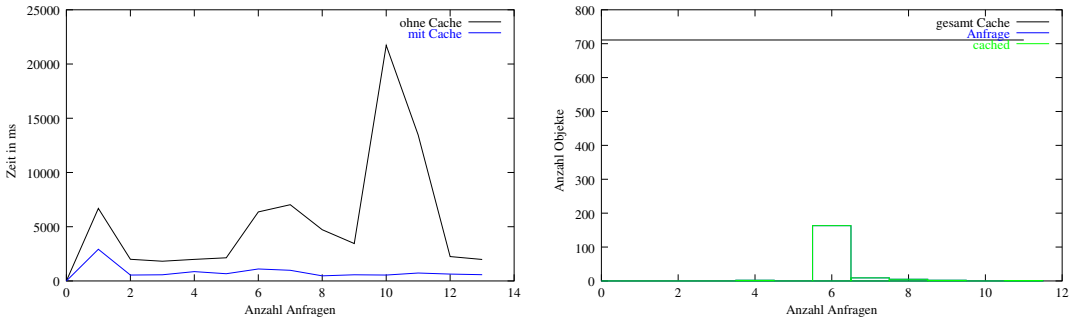


(a) Antwortzeit mit und ohne Cache

(b) Cache-Größe und Ergebnisanteil

Abbildung 20: Antwortzeiten und Ergebnisanteil zufälliger Anfragemix

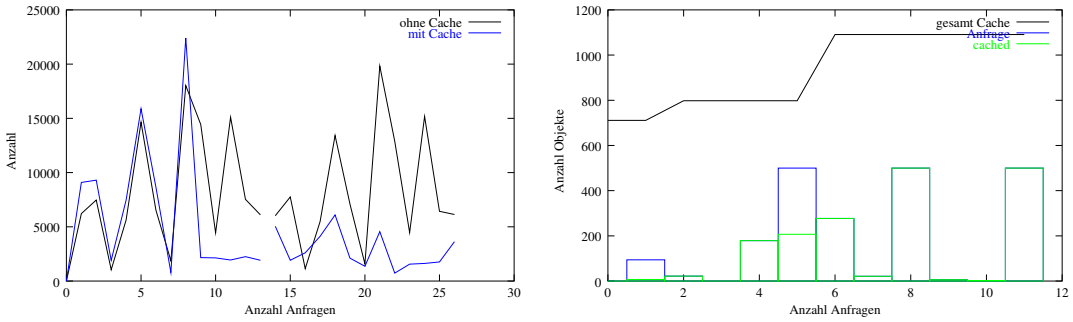
Der erste untersuchte Punkt ist der Beitrag des Cachings zur Reduzierung der Antwortzeit im Zeitablauf. Die Beschleunigung der Anfragebearbeitung wurde als Hauptanforderung an den realisierten Cache gestellt und findet deswegen hier die größte Beachtung. Um zur Analyse geeignete Werte zu erhalten, wurden die verschiedenen Anfragemixe wie zuvor beschrieben ausgeführt. Die Grafiken in den Abbildungen 20 bis 23 illustrieren jeweils die gemessene Antwortzeit mit (blaue Linie) und ohne (schwarze Linie) Cache-Unterstützung. Die zweite Teilgrafik stellt die Verbindung zu dem jeweils im Cache gefundenen Anteil der Ergebnisdaten (schwarz: Cache-Größe, blau: Umfang Ergebnismenge, grün: im Cache gespeicherter Teil des Ergebnisses) her.



(a) Antwortzeit mit und ohne Cache (b) Cache-Größe und Ergebnisanteil

Abbildung 21: Antwortzeiten und Ergebnisanteil einschränkender Anfragemix

Um das Verhalten des Caches in dem optimalen Fall, dass die angefragten Ergebnisdaten vollständig im Cache vorliegen, ebenfalls analysieren zu können, wurde jeder Anfragemix zweimal direkt nacheinander ausgeführt (außer der einschränkende Mix, bei dem keine neuen Daten im Cache aufgenommen werden). Spätestens bei der zweiten Ausführung sind die Daten im Cache komplett, denn die im ersten Lauf als Komplementärdaten erkannten Objekte werden in der Zwischenzeit bei den Quellsystemen angefragt und im Cache gespeichert. Der Punkt zwischen beiden Ausführungen wird durch einen kleinen Sprung in den Grafiken symbolisiert. Wird im Cache ein unvollständiges Ergebnis gefunden, so wird statt der gebildeten Komplementäranfrage immer die Originalanfrage zur Beantwortung an die Quellen gesandt. Das beruht zum Einen auf den in Abschnitt 5.4 erfassten Problemen bezüglich des Textähnlichkeitsoperators und ihrer Lösung, zum Anderen auch auf der Komplexität der entstehenden Komplementäranfrage. Dieser Punkt wird im Folgenden noch separat behandelt.

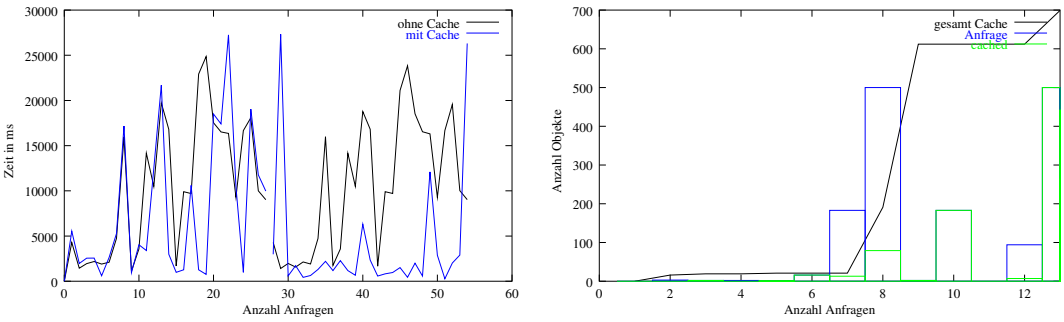


(a) Antwortzeit mit und ohne Cache (b) Cache-Größe und Ergebnisanteil

Abbildung 22: Antwortzeiten und Ergebnisanteil erweiternder Anfragemix

In allen Abbildungen ist zu erkennen, dass der Cache nach einer gewissen Anlaufphase die Antwortzeit deutlich reduzieren kann. Am Anfang der Testläufe passt sich die erzielte Antwortzeit an die Antwortzeiten der Quellsysteme an.

Nach der stetigen Füllung des Caches mit Daten wird gerade die Bearbeitung der einschränkenden und identischen Anfragen effizient unterstützt, da alle angefragten Daten im Cache vorliegen. Die stellenweise höhere Antwortzeit mit Cache-Unterstützung im Vergleich zur Zeit ohne Cache-Nutzung beruht auf dem zur Suche im Cache benötigten Mehraufwand und wird angesichts des Nutzens des Caches akzeptiert.

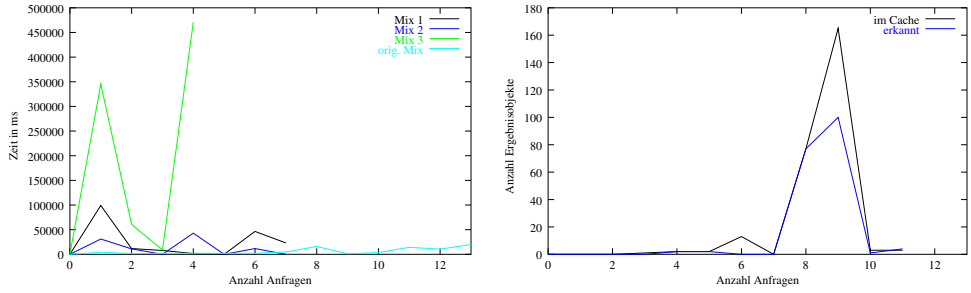


(a) Antwortzeit mit und ohne Cache

(b) Cache-Größe und Ergebnisanteil

Abbildung 23: Antwortzeiten und Ergebnisanteil großer Anfragemix

Die abgebildeten Resultate sind allerdings nur unter der Bedingung erzielbar, dass der '∼='-Operator korrekt unterstützt wird bzw. nicht genutzt wird. Wie Abschnitt 5.4 allerdings gezeigt hat, ist das mit der derzeitigen Realisierung nicht möglich. Die für jede Anfrage notwendige (alle Anfragen enthalten den '∼='-Operator) Originalanfrage kann nun auch parallel zur Suche im Cache bearbeitet werden. Die benötigte Zeit zur vollständigen Beantwortung gleicht sich dann den Anfragezeiten auf den Quellsystemen an. Entscheidend ist hier der Anteil des Ergebnisses, der bereits ohne Kommunikation an den Anwender übermittelt werden kann (in den Abbildungen 20 bis 23 jeweils grün dargestellt).



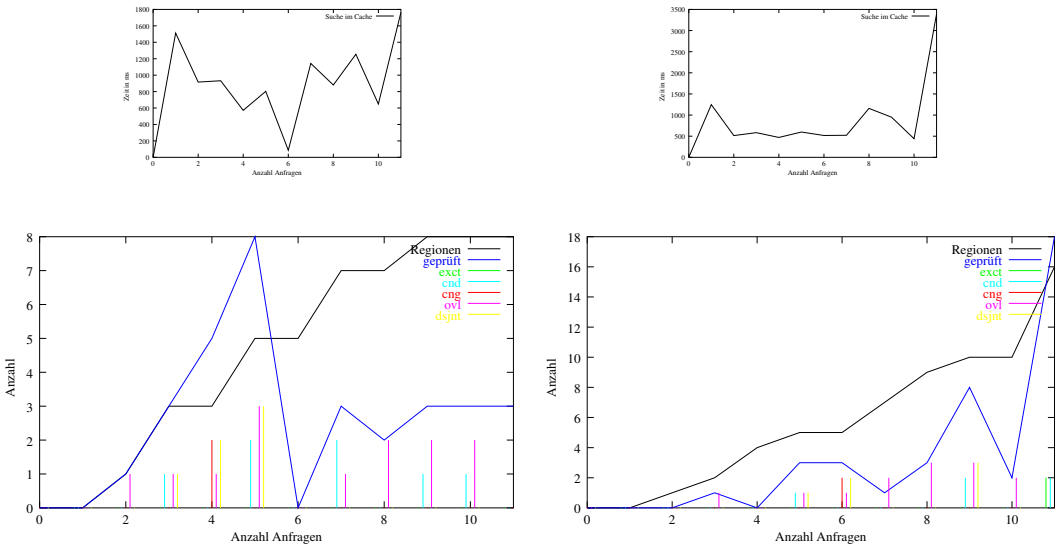
(a) Initiierte Komplementäranfragen

(b) Abweichungen vom Ergebnis

Abbildung 24: Probleme durch Textähnlichkeit und Komplementäranfrage

Die aus der Unterstützung des Ähnlichkeitsoperators rührende Abweichung der im Cache erkannten Daten von den tatsächlich gespeicherten war sehr ungenau zu ermitteln.

Die Anfragen lieferten zu wenig Ergebnisdaten, um aussagekräftige Werte zu erhalten. Die dennoch festgestellten Abweichungen sind in Abbildung 24(b) dargestellt (schwarz: Ergebnismenge, blau: erkannter Teil). Der zweite Teil der Abbildung 24 verdeutlich aufgetretene Probleme bei Beantwortung der Komplementäranfrage: Die erhaltenen Daten wären nicht nur unvollständig, die generierten Anfragen verursachen zudem eine wesentlich höhere Antwortzeit oder sind stellenweise gar nicht beantwortbar. Abgebildet sind die Antwortzeiten für die Komplementäranfragen, die bei der Bearbeitung der jeweiligen Anfragemixe gebildet wurden. Alle Folgen brechen an einem Punkt ab, an dem die Anfrage nicht weiter bearbeitbar ist. Die kleinste, einzige nicht abgebrochene Kurve, stellt die Antwortzeit der Quellen dar, die auch schon in den vorangegangenen Abbildungen integriert war. Die Ausführung der komplementären Anfrage ist um ein erhebliches Vielfaches langsamer. Aufgrund der beiden geschilderten Aspekte fällt eine Untersuchung hier somit nicht weiter in Betracht.

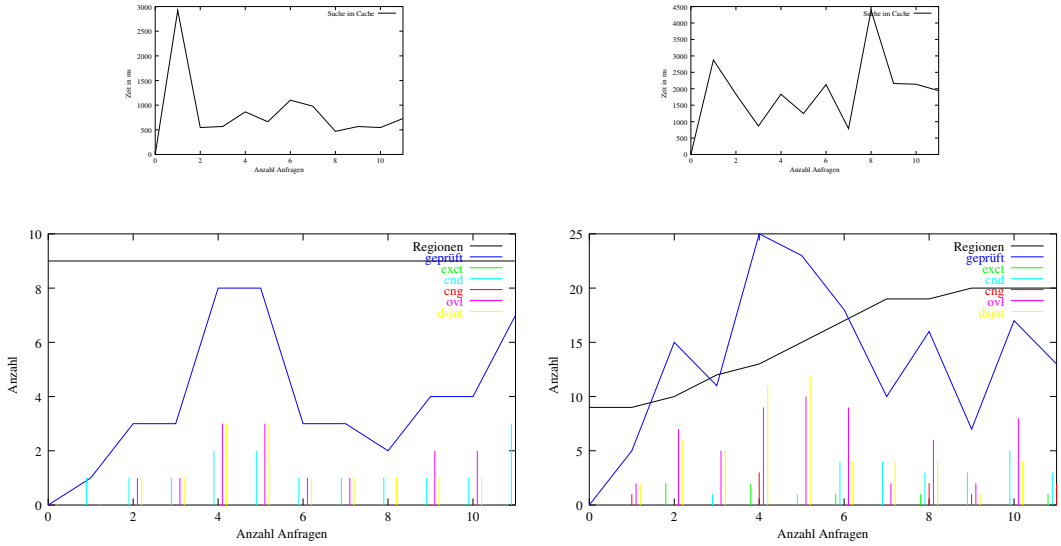


(a) normal (b) groß

Abbildung 25: Zufällige Anfragemixe

Der Aufwand zur Suche im Cache steigt mit der Anzahl im Cache erfasster Einträge merklich. Vor allem beim größeren Anfragemix sind deutliche Verzögerungen erkennbar. Diese beruhen größtenteils auf intensiven, teilweise noch ineffizient realisierten, Datenbankoperationen. Der benötigte Aufwand hängt allerdings nicht so stark wie erwartet von der Anzahl der in den Anfragen enthaltenen Teilziele ab. In den Abbildungen 25 und 26 werden die Anzahl untersuchter Einträge in Verbindung mit den auftretenden *match-types* und der Gesamtzahl im Cache enthaltener Regionen im Zeitablauf illustriert (blau: Anzahl von *match-type*-Aufrufen je Anfrage, schwarz: Anzahl Einträge im Cache, bunt: aufgetretene *match-types*). Die kleineren Abbildungen stellen die jeweils zur Suche im Cache benötigte Zeit dar. Wesentlicher Einflussfaktor für die benötigte Zeit ist hier die Anzahl zu untersuchender Cache-Einträge. Dies ist ein wesentlicher Faktor, allerdings nicht der einzig bedeutende.

So resultieren die im Zeitablauf der Anfragebearbeitung erkennbaren Spitzen der Kurve, die nicht in Verbindung mit einer hohen Anzahl zu prüfender Einträge zu bringen sind, aus benötigten Datenbankzugriffen und den geringen Hardwarekapazitäten des Testrechners.



(a) einschränkend (b) erweiternd

Abbildung 26: Einschränkender und erweiternder Anfragemix

Das vermehrte Auftreten von „overlapping match“- und „containing match“-Beziehungen (keine einzige Anfrage konnte ohne einen solchen *match-type* beantwortet werden) macht die Ausführung der Originalanfrage in jedem Schritt erforderlich, in dem die korrekte Unterstützung der Semantik von Operatoren nicht garantiert werden kann. Die Zusammenhänge zwischen der für die Suche benötigten und der Anzahl untersuchter Cache-Einträge, verleiht der Untersuchung einer hierarchischen Verwaltung der semantischen Regionen, wie sie in Abschnitt 4.3 skizzierte wurde, erneute Relevanz.

Die durchgeführten Tests und präsentierten Ergebnisse lassen, trotz ihres einfachen Charakters, Schlüsse auf das Leistungsverhalten des entwickelten Caches zu. Wenn Daten im Cache vorhanden sind, die der Anfrage entsprechen, so erlaubt die Nutzung des Caches eine wesentlich beschleunigte Übermittlung von Ergebnisdaten. Liegen die angefragten Daten komplett im Cache vor, was bei der Suche im YACOB-Mediator öfter auftreten kann, sind die erzielten Antwortzeiten nahezu optimal. Um den Cache optimal nutzen zu können, ist eine gewisse Anlaufzeit zum Füllen des Caches unabdingbar.

Die Probleme, die aus der Unterstützung des Textähnlichkeitsoperators rühren, beeinflussen den Beitrag des Caches zur Verringerung der Antwortzeit zwar, überdecken ihn aber nicht. Auch wenn in jedem Schritt die Originalanfrage ausgeführt wird und dies im Anschluss eine Duplikateliminierung nötig macht, können große Teile der Ergebnismenge bereits im Cache gefunden werden. Welche Auswirkungen der Operator tatsächlich auf das Verhalten des Caches hat, ist in detaillierteren Experimenten zu klären.

7 Schlussfolgerung und Ausblick

In dieser Arbeit wurde ein semantischer Cache für das YACOB-Mediatorsystem entwickelt. Der Cache ist eng an die im Mediator definierte Ontologie geknüpft. Die enge Bindung des Caches an die Konzepte führt zu einigen Besonderheiten und speziellen Möglichkeiten bei der Realisierung des Caches, die in den einzelnen Kapiteln der Arbeit detailliert behandelt worden. So hat diese enge Anbindung z.B. Einfluss auf die Bestimmung des *Query Containment* oder die Bildung der Komplementäranfrage. Im Cache werden XML-Daten und die zugehörigen XPath-Anfragen gespeichert. Die Identifikation von Cache-Einträgen und die Suche im Cache erfolgt nur anhand der entsprechenden XPath-Anfragen, die die Daten der Cache-Einträge semantisch beschreiben.

Der entwickelte Cache hat sich nach einer ersten Evaluation als durchaus nützlich erwiesen, um Kommunikationskosten zu reduzieren und vor allem um Antwortzeiten zu verringern. Gerade bei der Bedeutung der interaktiven Anfrageverfeinerung ist er von großem Nutzen: Ergebnismengen einschränkende Anfragen können sofort komplett aus dem Cache beantwortet werden. Bei Ergebnismengen erweiternden Anfragen ist zumindest der im Cache gespeicherte Teil der Daten schnell an den Anwender übermittelbar.

Diese ersten Ergebnisse müssen durch komplexe Simulationen und Experimente in möglichst realen Umgebungen bestätigt werden. Anpassungen zur Erhöhung der Effizienz des Vorgehens schließen Betrachtungen ein, wie die Menge der bei einer Suche zu bearbeitenden Cache-Einträge minimiert werden kann. Hauptaugenmerk könnte auf der Realisierung des skizzierten Verfahrens zur hierarchischen Verwaltung der semantischen Regionen liegen. Der zu erwartende wesentliche Vorteil ist die Möglichkeit der Einführung von logischen Verweisen zwischen den Regionen. Dadurch können auch Überlappungen ohne erheblichen Mehraufwand oder Datenredundanz erzielt werden. Ein weiterer Aspekt ist die Optimierung des internen Datenbankzugriffs, z.B. durch Nutzung von Indexen zur Unterstützung eines beschleunigten Zugriffs.

Als Problem hat sich die Unterstützung von CQuerys Textähnlichkeitsoperator erwiesen. Das vorgestellte Verfahren stellt dabei eine vorzeitige Lösung zum Umgang mit diesem Operator dar. Hier müssen bessere Strategien entwickelt werden. So könnten Informationen über Quelleneigenschaften genutzt werden. Dies kann sogar das Caching auf Quellenebene vorteilhafter werden lassen, als das hier realisierte Caching auf Konzeptebene. Trotz aller auftretenden Schwierigkeiten, konnte sich das Caching als geeignetes Mittel zur Verringerung der Antwortzeit bewähren.

Es bleiben eine ganz Reihe offener Aspekte bestehen, die in weiteren Beiträgen zum Thema des semantischen Cachings in ontologiebasierten Mediatoren behandelt werden können und sollten. Ein wesentlicher Punkt ist dabei z.B. die Entwicklung einer angepassten Strategie zur Ersetzung von Cache-Einträgen und zur Wahrung der Cache-Kohärenz. Dies umfasst auch die Analyse eines Wertes für die optimale Cache-Größe, unter anderem im Zusammenhang mit der Effizienz der Anfragebearbeitung. In diesem Zusammenhang ist auch die Verknüpfung der Komplementäranfrage mit einem guten Kohärenzprotokoll zu untersuchen. Da die Ausführung der originalen Anfrage eine mögliche Lösung für die aus dem Textähnlichkeitsoperator resultierenden Probleme ist, lassen sich diese Punkte in geeigneter Weise vereinen.

Andere nennenswerte Aspekte sind die Integration von komplexeren Anfragen, d.h. Anfragen die eventuell Attributverknüpfungen oder „höhere“ Operatoren enthalten, oder aber die fortgeschrittene Nutzung von semantischem Wissen. Das ist z.B. folgendermaßen zu verstehen: Angenommen, es ist bekannt, dass der Name „John“ nur männlichen Personen zugeordnet wird. Aus dieser Information kann man nun schließen, dass ein „John“-Objekt in der Ergebnismenge einer Anfrage nach allen männlichen Personen enthalten ist. Informationen dieser Art, die in diesem Fall sicher recht einfacher Natur sind, lassen sich mit den vorgestellten Methoden nicht ohne Modifikation erfassen. Auch die Bildung der Komplementäranfrage, ihre Beantwortbarkeit und eventueller Nutzen, der für ihre Generierung aus der Konzeptebene gewonnen werden kann, sind näher zu untersuchen.

Der Cache wurde bereits implementiert und kann in das YACOB-System integriert werden. Die Optimierung der erstellten Implementierung und ihre Einbindung in das YACOB-System, bis hin zur endgültigen Nutzung durch die Anwender, sind ebenfalls noch offene Punkte. Der Einfluss des Aufwands zur eigentlichen Suche im Cache ist zu minimieren. Schon Zustände, die der Cache nach recht kurzer Zeit annehmen kann, können schnell zu unerwünschten Verzögerungen in der Anfragebearbeitung führen. Wichtig ist dabei auch, den Zugriff auf die intern genutzte Datenbank XINDICE zu optimieren.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[ACPS96] Sibel Adali, K. Selcuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *SIGMOD Conference*, pages 137–148, 1996.

[CRS99] Boris Chidlovskii, Claudia Roncancio, and Marie-Louise Schneider. Optimizing Web Queries through Semantic Caching. In *Proc. 15emes Journees Bases de Donnees Avancees, BDA*, pages 23–40, 1999.

[CRZ03] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley Pearson Education, February 2003.

[DFJ+96] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB'96, Proc. of 22th Int. Conf. on Very Large Data Bases*, pages 330–341, Mumbai (Bombay), India, September 3–6 1996. Morgan Kaufmann.

[GG97] P. Godfrey and J. Gryz. Semantic Query Caching for Heterogeneous Databases. In *Intelligent Access to Heterogeneous Information, Proc. of the 4th Workshop KRDB-97, Athens, Greece*, volume 8 of *CEUR Workshop Proceedings*, pages 6.1–6.6, August 30 1997.

[GG99] P. Godfrey and J. Gryz. Answering Queries by Semantic Caches. In *Database and Expert Systems Applications, 10th Int. Conf., DEXA '99, Florence, Italy, Proc.*, volume 1677 of *LNCS*, pages 485 – 498. Springer, August 30 - September 3 1999.

[GSW96] Sha Guo, Wei Sun, and Mark Allen Weiss. Solving Satisfiability and Implication Problems in Database Systems. *TODS*, 21(2):270–293, 1996.

[Hal01] A. Y. Halevy. Answering Queries using Views: A Survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, December 2001.

[HS03] H. Höpfner and K.-U. Sattler. Towards Trie-Based Query Caching in Mobile DBS. In *Post-Proceedings of the Workshop "Scalability, Persistence, Transactions - Database Mechanisms for Mobile Applications", Karlsruhe, 10.-11. April 2003*, to appear 2003.

[KB96] A. M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal: Very Large Data Bases*, 5(1):35–47, January 1996.

[KMRU02] M. Klettke, H. Meyer, W. Retschitzegger, and R. Unland. Speicherung von XML-Dokumenten. In Rahm and Vossen [RV02], pages 135–162.

[KSGH03] M. Karnstedt, K.-U. Sattler, I. Geist, and H. Höpfner. Semantic Caching in Ontology-based Mediator Systems. In *Proceedings of the 3rd International*

Workshop of the GI Working Group "Web und Datenbanken", Berliner XML-Tage 2003, Berlin, 13. Oktober 2003, to appear 2003.

[LC98] D. Lee and W. W. Chu. Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Databases. In *CIKM'98 Workshop on Web Information and Data Management (WIDM'98), Washington, DC, USA*, November 6 1998.

[LC99] D. Lee and W. W. Chu. Semantic Caching via Query Matching for Web Sources. In *Proc. of the 1999 ACM CIKM Int. Conf. on Information and Knowledge Management, Kansas City, Missouri, USA*, pages 77–85. ACM, November 2–6 1999.

[LC01] D. Lee and W. W. Chu. Towards Intelligent Semantic Caching for Web Sources. *Journal of Intelligent Information Systems*, 17(1):23–45, November 2001.

[LHK01] Sang Ho Lee, Jin Seon Hong, and Larry Kerschberg. A Popularity-Driven Caching Scheme for Meta-search Engines: An Empirical Study. *Lecture Notes in Computer Science*, 2113:877+, 2001.

[LY85] Per-Åke Larson and H. Z. Yang. Computing Queries from Derived Relations. In Alain Pirotte and Yannis Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 259–269. Morgan Kaufmann, 1985.

[RD98] Q. Ren and M. Dunham. Semantic Caching and Query Processing, 1998.

[RD00] Q. Ren and M. Dunham. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In *Proc. of the 6th Annual Int. Conf. on Mobile Computing and Networking (MOBICOM-00)*, pages 210–242, New York, August 6–11 2000. ACM.

[Ren00] Q. Ren. Semantic Caching in Mobile Computing, February 2000.

[RI80] Daniel J. Rosenkrantz and Harry B. Hunt III. Processing Conjunctive Predicates and Queries. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 64–72. IEEE Computer Society, 1980.

[Rou91] Nick Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *TODS*, 16(3):535–563, 1991.

[RV02] E. Rahm and G. Vossen, editors. *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*. dpunkt.verlag, Heidelberg, September 2002.

[SCS02] K. Sattler, S. Conrad, and G. Saake. Datenintegration und Mediatoren. In Rahm and Vossen [RV02], pages 163–190.

[SGHS03] K. Sattler, I. Geist, R. Habrecht, and E. Schallehn. Konzeptbasierte Anfrageverarbeitung in Mediatorsystemen. In *Proc. BTW'03 - Datenbanksysteme für Business, Technologie und Web, Leipzig, 2003, GI-Edition, Lecture Notes in Informatics*, pages 78–97, 2003.

[SGS03] K. Sattler, I. Geist, and E. Schallehn. Concept-based Querying in Mediator Systems. Technical Report 2, Dept. of Computer Science, University of Magdeburg, 2003.

[Ull89] J. D. Ullman. *Principles of database and knowledge base systems, Volume 2*. W H Freeman & Co, 1989.

[Wei02] G. Weikum. Web Caching. In Rahm and Vossen [RV02], pages 191–216.

[Wie97] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 185–196. Morgan Kaufmann, San Francisco, CA, USA, 1997.