Informatik - Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Verteilte Metadatenverwaltung und Anfragebearbeitung für Internet-Datenquellen

Markus Keidl[1], Alfons Kemper[1], Donald Kossmann[2], Alexander Kreutz[1]

[1] Universität Passau, Fakultät für Mathematik und Informatik, D-94030 Passau, e-mail: ⟨*nachname*⟩@db.fmi.uni-passau.de
[2] Technische Universität München, Institut für Informatik, D-81667 München, e-mail: kossmann@in.tum.de

The date of receipt and acceptance will be inserted by the editor

Zusammenfassung Wir präsentieren in dieser Arbeit das ObjectGlobe-System, ein offenes und verteiltes Anfragebearbeitungssystem auf Internet-Datenquellen. ObjectGlobe erweitert die begrenzten Fähigkeiten des Internets bezüglich Anfragebearbeitung, indem es einen offenen Marktplatz schafft, in dem eine Vielzahl von Anbietern Rechenzeit, Daten und Anfrageoperatoren bereitstellen. Den Schwerpunkt dieses Beitrags bildet die im Rahmen des ObjectGlobe-Projekts entwickelte Metadatenverwaltung MDV, ein verteiltes Metadaten-Management-System. Es besitzt eine 3-schichtige Architektur und unterstützt Caching und Replikation in der Mittelschicht, so dass Anfragen lokal ausgewertet werden können. Benutzer spezifizieren die Informationen, die sie benötigen und die repliziert werden, mit Hilfe einer besonderen Regelsprache. Um Replikate aktuell zu halten und die Replikation von neuen und relevanten Informationen anzustoßen, verwendet das MDV-System einen neuartigen Publish&Subscribe-Algorithmus. Wir beschreiben diesen Algorithmus im Detail und zeigen wie er mit Hilfe eines normalen relationalen Datenbanksystems implementiert werden kann.

Schlüsselwörter Anfragebearbeitung, Integration verteilter Datenbanken, Metadatenverwaltung, Publish/Subscribe

Abstract In this work we present the ObjectGlobe system, an open and distributed query processing system for Internet data sources. ObjectGlobe extends the limited query processing capabilities of the Internet by creating an open market place where various suppliers provide computing time, data, and query operators. The main focus of this work is the distributed metadata management system MDV which was developed in the scope of the ObjectGlobe project. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally. Users specify the information they need and that is replicated using a specialized subscription language. In order to keep replicas up-to-date and initiate the replication of new and relevant information, MDV implements a novel publish&subscribe algorithm. We describe this algorithm in detail and show how it can be implemented using a standard relational database system.

Key words Query Processing, Integration of Distributed Databases, Metadata Management, Publish/Subscribe

CR Subject Classification: C.2.4, H.2.4, H.3.4

1 Einleitung

In der Vergangenheit wurde das Internet immer wieder mit einer riesigen, verteilten Datenbank verglichen. Hauptsächlich deswegen, weil es eine enorme Menge an Datenquellen zur Verfügung stellt. Ansonsten bietet es aber kaum Funktionalität zur Datenverarbeitung, wie man sie von modernen Datenbanksystemen gewohnt ist. Wir nehmen uns dieser Herausforderung im Rahmen von ObjectGlobe an, einem offenen und verteilten Anfragebearbeitungssystem, das die begrenzten Fähigkeiten des Internets bezüglich Anfragebearbeitung erweitert, indem es einen offenen Marktplatz für Daten und Anfrageoperatoren schafft.

Neben Datenquellen bietet das Internet aber auch eine Reihe anderer Ressourcen, die für die Anfragebearbeitung genutzt werden können, z. B. Rechenleistung und Speicherplatz. Allgemein kann man sagen, dass das Internet eine Vielzahl von Ressourcen zur Verfügung stellt, die von entsprechenden Anwendungen verwendet werden können. Elektronische Marktplätze und andere elektronische Dienste und Anwendungen greifen zur Erfüllung ihrer Aufgaben zunehmend auf diese Ressourcen zurück und erzeugen somit eine wachsende Nachfrage nach effektivem Ressourcenmanagement. Die dynamische Integration der Ressourcen und Dienste erfordert umfangreiche Metadaten zu ihrer Beschreibung und Verwaltung und zur Suche nach ihnen. Die Natur des Internets lässt solche Informationen sehr schnell altern. Außerdem müssen die Informationen für eine große Zahl von Benutzern und Anwendungen verfügbar sein. Kopien der Informationen müssen nahe bei den Benutzern gespeichert werden, die diese anwendungsspezifischen Informationen benötigen. Metadaten über diese Ressourcen und Dienste sind deshalb ein Schlüssel für den Erfolg derartiger Anwendungen.

Im Rahmen des ObjectGlobe-Projekts wurde deshalb die Metadatenverwaltung MDV entwickelt, ein verteiltes Metadaten-Management-System. MDV besitzt eine 3-Schichten-Architektur und unterstützt Caching und Replikation in der Mittelschicht, so dass Anfragen lokal ausgewertet werden können. Somit ist keine teure Kommunikation über das Internet notwendig, um Anfragen an die MDV zu stellen. Benutzer spezifizieren die Informationen, die sie benötigen und die repliziert werden, mit Hilfe einer eigenen Regelsprache. Dies reduziert die Datenmenge, die lokal angefragt werden muss, und verbessert dadurch die Performanz der Anfrageauswertung. Außerdem können damit Aktualisierungen der Metadatenbasis genau dorthin gesendet werden, wo sie benötigt werden. Dadurch wird die Größe der Aktualisierungsströme entscheidend eingeschränkt.

Das MDV-System implementiert einen neuartigen Publish&Subscribe-Algorithmus, um Replikate aktuell zu halten und die Replikation von neuen und relevanten Informationen zu initiieren. Wir beschreiben diesen Algorithmus im Detail, insbesondere wie er die möglicherweise enorme Menge von Abonnement-Regeln handhabt. Dazu werden zuerst sowohl die Metadaten als auch die Abonnement-Regeln in relationale Daten zerlegt. Der entscheidende Punkt des Algorithmus ist, dass diese Daten in einem normalen (kommerziellen) relationalen Datenbanksystem abgelegt werden. Dadurch wird die Bestimmung auszuwertender Regeln und die Auswertung selbst mit Hilfe einer Reihe von SQL-Anfragen möglich. Neben der Ausnutzung der Anfragefähigkeiten des Datenbanksystems ermöglicht die Verwendung eines normalen Datenbanksystems die Ausnutzung von dessen ausgereiften Fähigkeiten bezüglich Speicherung, Indexierung und Skalierbarkeit.

Der Rest dieser Arbeit ist wie folgt gegliedert: Abschnitt 2 gibt einen Überblick über das ObjectGlobe-System. Abschnitt 3 präsentiert die Architektur der Metadatenverwaltung MDV und Abschnitt 4 beschreibt den Publish&Subscribe-Algorithmus. Abschnitt 5 gibt einen Überblick über verwandte Arbeiten und Abschnitt 6 beschließt diese Arbeit mit einer Zusammenfassung und einem Ausblick.

2 Das ObjectGlobe-System

Ziel des ObjectGlobe-Projekts ist die Entwicklung eines Systems, mit dem jede beliebige Art von Funktionalität für die Anfragebearbeitung im Internet verteilt werden kann. Unser System erlaubt die ad-hoc Komposition von beliebigen Diensten für die Anfragebearbeitung, die im Internet in einer Art offenem Markt angeboten werden. Wir unterscheiden drei Arten von Anbietern (*Providern*) spezialisierter Dienste: *Data-Provider* bieten Daten an, *Function-Provider* stellen Anfrageoperatoren zur Verfügung, mit denen Daten verarbeitet werden können, und *Cycle-Provider* stellen Rechenzeit zur Bearbeitung von Anfragen zur Verfügung. Es ist zwar möglich (und sogar wahrscheinlich), dass ein einzelner Rechner alle drei Arten von Diensten anbietet, für den Rest dieser Arbeit werden die Provider aber als getrennte Einheiten betrachtet. Eine ausführliche Beschreibung des ObjectGlobe-Systems ist in [7] zu finden. Eine Anwendung von ObjectGlobe im Bereich der Realisierung skalierbarer elektronischer Marktplätze wurde in [26] beschrieben.

2.1 Anfragebearbeitung in ObjectGlobe

Die Auswertung einer Anfrage erfordert in ObjectGlobe vier Schritte:

1. Lookup: In dieser Phase werden von der MDV Metadaten über alle Datenquellen, Cycle-Provider und Operatoren angefordert, die bei der Bearbeitung der Anfrage nützlich sein könnten. Zusätzlich stellt die MDV Sicherheitsanforderungen zur Verfügung. Diese legen die Ressourcen fest, auf die der Benutzer Zugriff hat, der die Anfrage gestartet hat.
2. Optimize: Ein kostenbasierter Optimierer erzeugt aus den Informationen, die im Lookup-Schritt bestimmt wurden, einen gültigen Anfrageplan (soweit es die bekannten Sicherheitsanforderungen betrifft). Der Plan ist mit Informationen annotiert, welcher Operator auf welchem Cycle-Provider ausgeführt werden soll und von welchem Function-Provider externe Operatoren geladen werden sollen.
3. Plug: Der Anfrageplan wird an alle im Plan aufgeführten Cycle-Provider verteilt. Die externen Operatoren werden von den Function-Providern geladen und instanziiert. Außerdem werden die Kommunikationsverbindungen (d. h. Sockets) aufgebaut. Wenn gewünscht, werden Daten verschlüsselt und authentifiziert übertragen.
4. Execute: Der Plan wird entsprechend einem Iteratormodell [19] ausgeführt. Zusätzlich zu den externen Operatoren, die von Function-Providern geladen werden, stellt ObjectGlobe interne Operatoren zu Verfügung, etwa für Selektion, Projektion, Join, Union, Nesting, Unnesting, Senden und Empfangen von Daten. Die Ausführung des Plans wird ständig überwacht, sodass auf Fehler reagiert werden kann, indem der Plan angehalten und notfalls abgebrochen wird.

Das ObjectGlobe-System ist aus zwei Gründen in Java implementiert: Zum einen ist Java plattformunabhängig, sodass ObjectGlobe auf verschiedensten Rechnern mit wenig Aufwand installiert werden kann. Zum anderen bietet Java die Funktionalität, um ObjectGlobe dynamisch und auf sichere Art und Weise erweitern zu können.

Zur Illustration der Anfragebearbeitung in ObjectGlobe dient das Beispiel aus Abbildung 1. Es zeigt zwei Data-Provider, A und B, und einen Function-Provider. Wir nehmen an, dass die Data-Provider zugleich Cycle-Provider sind, so dass auf den Rechnern von A und B das ObjectGlobe-System installiert ist. Außerdem agiert der Klient in diesem Beispiel als Cycle-Provider. Data-Provider A stellt zwei Datenquellen

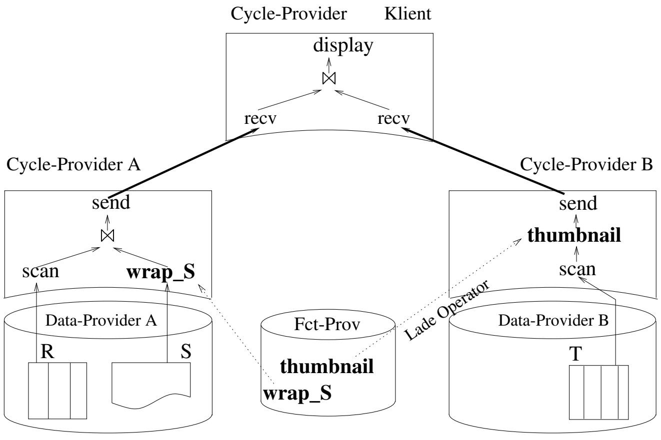

Abbildung 1 Verteilte Anfragebearbeitung in ObjectGlobe

zur Verfügung, eine relationale Tabelle R und eine Datenquelle S, die ein Wrapper in ein für die Anfragebearbeitung passendes Format umwandelt. Data-Provider B besitzt eine (geschachtelt) relationale Tabelle T. Der Function-Provider bietet zwei Anfrageoperatoren an: einen Wrapper (*wrap_S*), um S in das interne Datenformat des Anfrageprozessors (ein geschachtelt relationales Format) umzuwandeln, und einen Kompressionsalgorithmus (*thumbnail*), der z. B. auf ein Bildattribut von T angewendet werden kann.

2.2 Lookup-Service und Optimierung

Der Lookup-Service übernimmt in ObjectGlobe die Rolle, die ein Katalog oder eine Metadatenverwaltung in einem traditionellen Anfragebearbeitungssystem einnimmt. Provider werden registriert, bevor sie in ObjectGlobe eingesetzt werden können. Auf diese Weise werden die Informationen über verfügbare Dienste, entsprechend den Anforderungen, Schritt für Schritt erweitert. Das ObjectGlobe-System verwendet die Metadatenverwaltung MDV als Lookup-Service.

Die Metadaten über Provider sind sehr detailiert, da sie die einzigen Informationen über Dienste sind, die der Optimierer zur Erzeugung eines gültigen Auswertungsplans erhält. Die Metadaten enthalten z. B. Informationen über Namen und Signaturen von Operatoren (Ähnliche Metadaten beschreibt die Web Service Description Language (WSDL) [11] im Bereich Web Services.), Kostenmodelle von Operatoren, Leistungsdaten von Cycle-Providern sowie Schema-Informationen und Statistiken über Datenquellen von Data-Providern. Zusätzlich werden Autorisierungsdaten über Dienste miteinbezogen. Damit werden während des Optimierungsprozesses Kompatibilitätsmatrizen erzeugt. Diese enthalten Informationen über erlaubte Kombinationen von Diensten, die möglicherweise an einer bestimmen Stelle im Anfrageauswertungsplan an der Anfragebearbeitung beteiligt sind. Aufgrund dieser Autorisierungseinschränkungen kann es passieren, dass der Optimierer keinen Anfrageauswertungsplan findet, obwohl der Lookup-Service notwendige Dienste geliefert hat.

Der Optimierer verwendet dynamische Programmierung nach Art des System-R-Optimierers, um alternative Auswertungspläne aufzuzählen. D. h. Pläne werden von unten nach oben aufgebaut: Zuerst werden so genannte *Zugriffspläne* erzeugt. Diese legen fest, wie die Datenquellen gelesen werden (d. h. auf welchem Cycle-Provider und mit welchem *scan*- oder *wrapper*-Operator). Anschließend werden *Join-Pläne* erzeugt, basierend auf den Zugriffsplänen und (später) den einfacheren Join-Plänen. Offensichtlich kann der Suchraum für komplexe ObjectGlobe-Anfragen zu groß für vollständige dynamische Programmierung werden. Um derartige Anfragen trotzdem optimieren zu können, haben wir eine Erweiterung namens *Iterative Dynamische Programmierung (IDP)* entwickelt. IDP ist adaptiv; sie beginnt wie dynamische Programmierung und sie verhält sich auch genau so, wenn die Anfrage einfach genug ist. Sollte sich die Anfrage als zu komplex herausstellen, verwendet IDP Heuristiken, um einen akzeptablen Plan zu finden. Details und eine vollständige Analyse von IDP gibt [28].

2.3 Quality of Service (QoS)

Das ObjectGlobe-System kann aus einer großen Zahl von Cycle-Providern und noch weit mehr Data-Providern bestehen, so dass trotz der Optimierungen aus 2.2 Pläne entstehen können, die mehr Rechenzeit und mehr Kosten verbrauchen, als ein Benutzer zu investieren bereit ist. Ein derartig offenes System muss es einem Benutzer deshalb ermöglichen, Qualitätskriterien für die Bearbeitung selbst festzulegen. Diese Kriterien können in drei Bereiche aufgeteilt werden:

Ergebnis: Benutzer wollen möglicherweise die Größe des Ergebnisses einer Anfrage nach oben oder unten beschränken. Das Einschränken der Datenmenge, die für die Beantwortung der Anfrage verwendet wird, und ihrer Aktualität können dazu eingesetzt werden, ein Ergebnis zu erhalten, das auf einer aktuellen und ausreichend großen Teilmenge der verfügbaren Daten basiert.

Kosten: In unserem Szenario können Provider ihre Dienste in Rechnung stellen, also muss ein Benutzer eine obere Grenze für die Bearbeitungskosten seiner Anfragen festlegen können.

Zeit: Antwortzeit ist ein weiteres, wichtiges Qualitätskriterium bei der interaktiven Anfragebearbeitung. Benutzer können zum einen an einer schnellen Produktion des ersten Antworttupels interessiert sein. Sie können aber auch an einer möglichst schnellen Ausführung der kompletten Anfrage Interesse haben. Die schnelle Produktion der ersten Tupel kann dazu wichtig sein, dass der Benutzer diese Tupel betrachten kann, während der Rest der Anfrage im Hintergrund berechnet wird.

In vielen Fällen werden nicht alle QoS-Parameter wichtig sein. Wie auch in Echtzeitsystemen können einige Bedingungen strikt (oder hart) sein, andere können gelockert werden. Eine detailierte Beschreibung der QoS-Eigenschaften von ObjectGlobe sowie ein Vergleich mit existierenden Ansätzen finden sich in [6].

Die Bearbeitung einer Anfrage beginnt in ObjectGlobe bei der Formulierung der Anfrage selbst. Hier kann ein

Benutzer QoS-Kriterien und Statistiken über die Ressourcen (Provider und Kommunikationsverbindungen) festlegen. QoS-Kriterien spielen in allen Phasen der Anfragebearbeitung eine Rolle. Zuerst generiert der Optimierer einen Auswertungsplan, dessen geschätzte Qualitätsparameter die vom Benutzer für die Anfrage festgelegten Qualitätskriterien voraussichtlich erfüllen. Der Optimierer legt für jeden Teilplan minimale Qualitätskriterien an, die einzuhalten sind, damit die Qualitätsabschätzungen des ausgewählten Gesamt-Plans erfüllt werden können. Außerdem wird auch der Ressourcen-Bedarf angegeben, der für die Erfüllung dieser Qualitätskriterien notwendig ist. Sollte der Bedarf während der Plug-Phase mit den verfügbaren Ressourcen nicht befriedigt werden können, wird der Plan angepasst oder abgebrochen. Das QoS-Management reagiert genauso, wenn entsprechende Monitor-Komponenten während der Anfragebearbeitung eine mögliche Verletzung der QoS-Bedingungen vorhersagen.

2.4 Sicherheit und Datenschutz

Sicherheit ist für den Erfolg eines offenen und verteilten Systems wie ObjectGlobe naheliegenderweise ein entscheidender Punkt. Welche speziellen Sicherheitsaspekte dabei von Interesse sind, hängt vom jeweiligen Standpunkt ab. Einerseits brauchen Data- und Cycle-Provider ein leistungsfähiges Sicherheitssystem, um ihre Ressourcen gegen unautorisierte Zugriffe und Angriffe von böswilligem Code zu schützen, z. B. gegen Denial-of-Service-Angriffe. Außerdem haben diese Provider ein legitimes Interesse an der Identität ihrer Benutzer, etwa zwecks Autorisierung. Benutzer andererseits möchten sich über die Semantik eines externen Operators sicher sein, damit sie sich auf das Ergebnis einer Anfrage verlassen können. Dazu muss auch die Kommunikation vor Manipulationen geschützt werden. Desweiteren sind Benutzer auch an Datenschutz interessiert, d. h. anderen Personen dürfen vertrauliche Daten nicht lesen oder mithören können. Wir skizzieren im Folgenden unser Sicherheitskonzept für ObjectGlobe, wobei wir für weitergehende Informationen auf [39] verweisen. Die Sicherheitsmaßnahmen werden entsprechend ihrem Anwendungszeitpunkt eingeteilt:

Präventive Maßnahmen Präventive Maßnahmen finden statt, bevor ein Operator in Anfragen verwendet wird. Sie beinhalten die Überprüfung der Ergebnisse eines Operators, Belastungstests und Validierung des Kostenmodells. Diese Kontrollen führen vertrauenswürdige Drittanbieter durch. Sie erstellen ein Dokument mit den Untersuchungsergebnissen des getesteten Operators und signieren es digital. Zur Unterstützung dieser Kontrollen haben wir einen Validierungsserver entwickelt, der semi-automatisch Testdaten generiert, einen Operator ausführt und das Ergebnis der Ausführung mit dem Ergebnis vergleicht, das er mit Hilfe einer ausführbaren formalen Spezifikation oder einer Referenzimplementierung erhalten hat. Zusätzlich überprüft der Validierungsserver, ob die Ausführungskosten innerhalb der durch das Kostenmodell festgelegten Grenzen liegen.

Ziel dieser präventiven Maßnahmen ist die Stärkung des Vertrauens in das gutartige Verhalten von externen Operatoren. Benutzer, die großen Wert auf Sicherheit legen, werden exklusiv zertifizierte externe Operatoren verwenden, um zu gewährleisten, dass alle Operatoren ihr Ergebnis entsprechend der angegebenen Semantik berechnen.

Überprüfungen während der Planverteilung Während der Planverteilung ist das Sicherheitssystem für drei Aufgaben zuständig: Aufbau sicherer Kommunikationsverbindungen, Authentifizierung und Autorisierung. ObjectGlobe unterstützt die bekannten Standards SSL [18] und/oder TLS [14] zur Verschlüsselung und digitalen Signierung von Nachrichten. Beide Protokolle ermöglichen die Authentifizierung der Kommunikationspartner mit Hilfe von X.509-Zertifikaten [23]. Wenn Benutzer Pläne digital signieren, werden solche Zertifikate auch zur Authentifizierung verwendet. Zusätzlich unterstützt ObjectGlobe die Einbettung von verschlüsselten Passwörtern in den Anfrageplan. Diese Passwörter können Wrapper zur Passwort-basierten Authentifizierung bei Legacy-Systemen verwenden. Natürlich können Benutzer anonym bleiben, wenn sie öffentlich benutzbare Ressourcen verwenden.

Provider entscheiden autonom, basierend auf der Identität der Benutzer, ob der Benutzer autorisiert ist, einen Operator auszuführen, auf Daten zuzugreifen oder einen externen Operator zu laden. Provider können (müssen aber nicht) den Zugriff auf oder die Benutzung ihrer Ressourcen auf bestimmte Benutzergruppen einschränken. Zusätzlich können sie den Informationsfluss (bzw. Funktionscodefluss) einschränken, um zu gewährleisten, dass nur – aus ihrer Sicht – vertrauenswürdige Cycle-Provider in der Anfragebearbeitung verwendet werden. Dem Lookup-Service von ObjectGlobe müssen diese Sicherheitsanforderungen bekannt sein, damit gültige Auswertungspläne erzeugt werden können und Fehler zur Ausführungszeit vermieden werden.

Laufzeit-Maßnahmen Zur Verhinderung von böswilligen Aktionen externer Operatoren greift ObjectGlobe auf die Sicherheitsinfrastruktur von Java zurück. Externe Operatoren werden isoliert und in geschützten Bereichen, so genannten *Sandkästen*, ausgeführt. Cycle-Provider können dadurch z. B. verbieten, dass externe Operatoren auf das Dateisystem zugreifen. Weiterhin wird dadurch verhindert, dass externe Operatoren Daten nach außen geben, z. B. über Netzwerkverbindungen.

Zusätzlich kann eine Monitor-Komponente zur Laufzeit gegen Denial-of-Service-Angriffe vorgehen. Dazu greift sie auf die Kostenmodelle der Operatoren zurück und Überwacht den Ressourcenverbrauch der Operatoren (z. B. Speicherverbrauch und Prozessorzyklen). Wenn ein Operator mehr Ressourcen verbraucht als das Kostenmodell vorhersagt, kann er abgebrochen werden.

```
<CycleProvider rdf:ID="host">
  <serverHost>
    pirates.uni-passau.de
  </serverHost>
  <serverPort>5874</serverPort>
  <serverInformation>
    <ServerInformation rdf:ID="info"
       memory="92" cpu="600" />
  </serverInformation>
</CycleProvider>
```

Abbildung 2 Auszug aus einem MDV RDF-Dokument

3 Die Metadatenverwaltung MDV

In diesem Abschnitt beschreiben wir die Metadatenverwaltung MDV, die zur Realisierung des Lookup-Dienstes von ObjectGlobe verwendet wird. Die wichtigsten Eigenschaften des MDV-Systems sind: eine 3-Schichten-Architektur, die eine flexible Anpassung an wechselnde Lastsituationen erleichtert, ein effizienter Zugriff auf Metadaten durch Caching und ein Publish&Subscribe-Mechanismus zur Erhaltung der Cache-Konsistenz.

Unser System verwendet RDF [29] als Datenmodell (mit der XML-Syntax für RDF-Dokumente) und RDF-Schema [8] zur Festlegung des Schemas, dem die RDF-Daten entsprechen müssen.

3.1 Ein Beispiel

Abbildung 2 zeigt einen Auszug aus einem RDF-Dokument `doc.rdf`, in dem die zwei Ressourcen `CycleProvider` und `ServerInformation` definiert werden. Erstere repräsentiert einen Server im Internet, der beliebige ObjectGlobe-Dienste ausführen kann, letztere enthält Informationen über den Rechner, auf dem dieser Server läuft. Die Property `rdf:ID` legt für eine Ressource einen lokalen Identifikator fest, im Beispiel `host` und `info`. Ein eindeutiger Identifikator, *URI-Referenz* genannt, entsteht durch die Kombination des lokalen Identifikators mit der (global eindeutigen) URI des Dokuments. Die CycleProvider-Ressource enthält drei weitere Properties: `serverHost`, die den DNS-Namen des Provider-Rechners festlegt, `serverPort`, welche den Port des Providers definiert, und `serverInformation`, eine Referenz auf eine ServerInformation-Ressource. Diese enthält die Größe des Hauptspeichers des Provider-Rechners in MB (Property `memory`) und die Geschwindigkeit der CPU in MHz (Property `cpu`). Properties (wie `serverInformation`) referenzieren Ressourcen immer über deren URI-Referenz, d. h. RDF unterscheidet nicht zwischen geschachtelten und referenzierten Ressourcen. Es ist also unerheblich, ob Ressourcen als geschachtelte Elemente definiert werden (wie in Abbildung 2) oder sonstwo im selben oder sogar einem anderen Dokument.

3.2 Architekturüberblick

Abbildung 3 zeigt die hierarchische, 3-schichtige Architektur des MDV-Systems, bestehend aus *öffentlichen* und *lokalen Metadatenverwaltungen* sowie *MDV Klienten*.

Öffentliche Metadatenverwaltungen (Ö-MDVs), auch als MDV-Backbone bezeichnet, sind über das ganze Internet verteilt, um einen gleichartigen Zugriff zu ermöglichen, bezogen auf Latenzzeit und Metadaten-Inhalt. Letzteres wird erreicht durch die Verwendung des selben Schemas und die Replikation der Metadaten untereinander. Im Grunde bildet der Backbone ein verteiltes Datenbanksystem mit flacher Hierarchie, vollständiger Synchronisation und Replikation.[1] Metadaten, die in Ö-MDVs gespeichert werden, gelten als global und öffentlich zugänglich.

Lokale Metadatenverwaltungen (L-MDVs) sind diejenigen Komponenten, die die tatsächliche Metadaten-Anfragebearbeitung übernehmen. Aus Effizienzgründen, d. h. um Kommunikation über das Internet zu vermeiden, puffern L-MDVs globale Metadaten und verwenden ausschließlich lokal verfügbare Metadaten für die Anfragebearbeitung. L-MDVs sollten sich deshalb nahe bei den Anwendungen befinden, die auf sie zugreifen, z. B. im selben LAN. Der Cache einer L-MDV sollte Metadaten enthalten, die für die Benutzer oder Anwendungen, die auf sie zugreifen, relevant sind. L-MDVs benutzen einen Publish&Subscribe-Mechanismus, um alle relevanten Metadaten von einer Ö-MDV zu holen und Änderungen dieser Daten zu empfangen, d. h. um ihren Cache konsistent zu halten. Wenn eine L-MDV Daten einer Ö-MDV abonniert (*Subscribe*-Schritt), registriert sie einen Satz von Abonnement-Regeln, die diejenigen Metadaten beschreiben, welche die L-MDV puffern will. Ö-MDVs benutzen die Abonnement-Regeln, um neue Metadaten sowie Änderungen und Löschungen an L-MDVs zu publizieren (*Publish*-Schritt). Zusätzlich zu globalen Metadaten speichern L-MDVs auch lokale Metadaten, die nicht der Öffentlichkeit zugänglich sein sollen und deshalb nicht an den Backbone weitergeleitet werden. Lokale Metadaten müssen bei der Registrierung explizit als solche gekennzeichnet werden.

Anwendungen und Benutzer, die auf das MDV-System zugreifen, werden als *MDV-Klienten* bezeichnet. Sie können Metadaten an einer L-MDV anfragen, indem sie eine (deklarative) Anfragesprache verwenden, die aus Platzgründen nicht in diesem Beitrag vorgestellt wird. Die Sprache gleicht allerdings der Regelsprache, die in Abschnitt 3.3 erläutert wird. Benutzer können außerdem Metadaten an einer Ö-MDV durchsuchen (wie in Abbildung 3 dargestellt) und zum Caching auswählen. Ihre L-MDV wird passende Regeln erzeugen und ihren Satz an Abonnement-Regeln entsprechend anpassen.

Die Administrierung von Metadaten, d. h. die Registrierung von neuen Metadaten, sowie das Aktualisieren und Löschen von existierenden Metadaten, erfolgt an den Ö-MDVs. Neue Metadaten müssen in einem gültigen RDF-Dokument registriert werden; die Aktualisierung von Metadaten erfolgt durch erneute Registrierung der aktualisierten Version eines

[1] Natürlich ist eine erweiterte verteilte Architektur bezüglich Partitionierung und Replikation möglich. Das ist aber nicht Fokus unserer Arbeit. Für Arbeiten über Partitionierung und Replikation in verteilten Datenbanksystemen siehe [34].

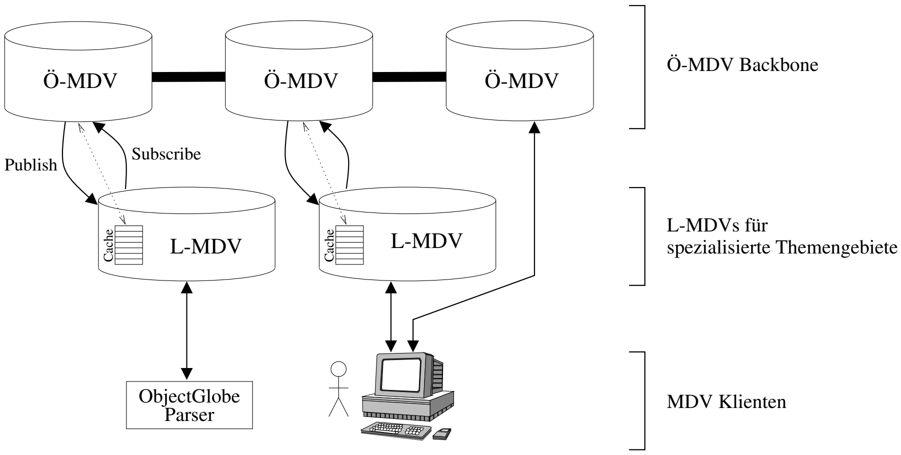

Abbildung 3 Überblick über die Architektur des MDV-Systems

bereits registrierten Dokuments. Löschen kann durch die Entfernung von Teilen des Dokuments und anschließende Aktualisierung des Dokuments geschehen oder durch die Entfernung des kompletten Dokuments mit dem gesamten Inhalt. Nur auf diese Weise können Metadaten hinzugefügt, aktualisiert und gelöscht werden. Die Anfragesprache der MDV stellt keine entsprechende Funktionalität bereit.

Das MDV-System ist in Java implementiert, so dass es portierbar ist und mit geringem Aufwand installiert werden kann. Es setzt auf ein relationales Datenbanksystem als Datenspeicher auf, wobei jede Ö-MDV und jede L-MDV jeweils eine separate relationale Datenbank verwenden, d. h. die als Cache agierenden L-MDVs werden genauso Datenbank gestützt implementiert wie der MDV-Backbone aus Ö-MDVs. RDF-Dokumente werden wie in [16] beschrieben auf Tabellen abgebildet. Suchanfragen werden in SQL-Joinanfragen umgewandelt. Die Umwandlung von Suchanfragen in SQL-Anfragen ist relativ komplex und die Beschreibung aller Details liegt jenseits des Rahmens dieses Beitrags.

3.3 Das Publish&Subscribe-System

Abonnement-Regeln einer L-MDV werden von Benutzern festgelegt, während sie Metadaten durchsuchen und auswählen, oder vom Administrator. Regeln müssen die Art der Metadaten beschreiben, an denen eine L-MDV interessiert ist, weil nur Metadaten, die diesen Regeln entsprechen, lokal gepuffert und für die Anfragebearbeitung benutzt werden. Eine Regel wird (informell) entsprechend folgender, SQL-ähnlicher Syntax definiert:

search $Extension$ e register e
where $Predicates(e)$

Die Regel wählt alle Ressourcen e aus, die ein Element der Ausprägung *Extension* sind – wobei es sich entweder um eine Klasse aus dem Schema oder eine andere Abonnement-Regel handeln kann – und den `where`-Teil der Regel erfüllen. *Predicates* ist eine Konjunktion von elementaren Prädikaten, welche die Form $X \circ Y$ haben, wobei X und Y entweder Konstanten oder gültige Pfadausdrücke (gemäß dem Schema) sind und $\circ \in \{ =, !=, <, <=, >, >=, \texttt{contains}\}$. Das MDV-System stellt einen speziellen *any*-Operator `?` zur Verfügung, der auf mengenwertige Properties angewendet werden kann.[2] Die derzeitige Implementierung unterstützt kein `OR`, eine Regel mit `OR`-Operator kann mit der Bool'schen Algebra aber leicht in mehrere Regeln aufgeteilt werden, die kein `OR` enthalten.

Beispiel 1 Die folgende Regel selektiert alle Ressourcen, die eine Instanz der im Schema definierten Klasse CycleProvider sind, eine Property `serverHost` besitzen, welche die Zeichenkette 'uni-passau.de' enthält, und deren Property `serverInformation` eine ServerInformation-Ressource mit einem `memory`-Wert größer als 64 referenziert:

```
search     CycleProvider c
register   c
where      c.serverHost contains 'uni-passau.de' and
           c.serverInformation.memory > 64
```

Diese Regel wählt Metadaten also alle CycleProvider aus, die auf einem Rechner in der Domain 'uni-passau.de' laufen, der mehr als 64MB Hauptspeicher besitzt. Die Cycle-Provider-Ressource in Abbildung 2 etwa entspricht dieser Regel.

4 Der Publish&Subscribe-Algorithmus

In diesem Abschnitt beschreiben wir unseren Publish&Subscribe-Algorithmus, insbesondere eine seiner Kernkomponenten, den Filter-Algorithmus. Eine wichtige Herausforderung in einem Publish&Subscribe-System ist die Auswertung der Abonnement-Regeln. Sie ist notwendig, um alle Abonnenten zu bestimmen, die über neue, aktualisierte oder gelöschte Daten benachrichtigt werden müssen. Um die Auswertung der möglicherweise riesigen Menge aller Abonnement-Regeln zu vermeiden, bestimmt der Filter-Algorithmus eine (kleine) Teilmenge von Regeln, die höchstens von der Änderung der Daten betroffen sind. Zusätzlich nutzt der Filter Regel- und Prädikat-Redundanzen aus und wertet die betroffen Regeln inkrementell aus.

[2] Bei mengenwertigen Properties müssen alle Elemente denselben Typ besitzen, auch wenn RDF dies formal nicht erzwingt.

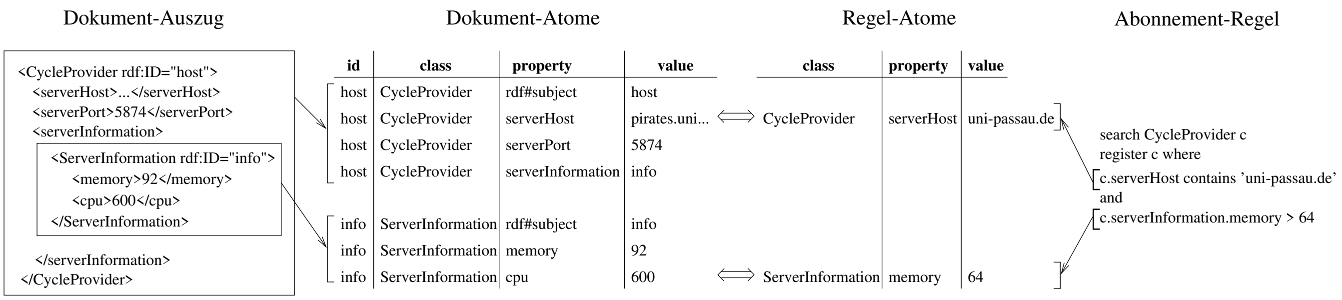

Abbildung 4 Grundsätzliche Idee des Filter-Algorithmus

Einer der wichtigsten Vorteile des Filter-Algorithmus ist, dass er auf normaler, relationaler Datenbank-Technologie basiert. Sowohl die Metadaten, d. h. die RDF-Dokumente, als auch die Abonnement-Regeln werden als relationale Daten, d. h. als Tupel, im Datenbanksystem gespeichert. Die Skalierbarkeit moderner Datenbanksysteme ermöglicht die effiziente Speicherung sehr großer Datenmengen, also auch sehr vieler Regeln. Die Auswertung sämtlicher Regeln auf neuen, geänderten oder gelöschten Daten erfolgt durch eine Reihe von (generierten) SQL-Anweisungen. Der Prototyp, den wir basierend auf dem MDV-System, seiner Regelsprache und dem RDF-Datenmodell entwickelt haben, setzt diese Anfragen per JDBC an die Datenbank ab. Da Metadaten und Regeln als Daten im Datenbanksystem gespeichert und normale SQL-Anfragen zur Auswertung verwendet werden, kann der Prototyp die Massenverarbeitungsmechanismen des Datenbanksystems voll ausnutzen. Zusätzlich kann er solche Datenbank-Vorteile wie Indexierbarkeit der Daten, Robustheit und Skalierbarkeit ausnutzen.

4.1 Überblick über den Ansatz

Man betrachte folgende Regel, die alle Cycle-Provider (d. h. ihre Ressourcen) in der Domain 'uni-passau.de' auswählt:

`search` CycleProvider c `register` c
`where` c.serverHost `contains` 'uni-passau.de'

Die Regel muss ausgewertet werden, wenn eine Ressource der Klasse CycleProvider registriert, aktualisiert oder gelöscht wird. Folgende Regel, die alle Cycle-Provider auswählt, die auf einem Rechner mit mehr als 64MB Hauptspeicher laufen, zeigt, dass es nicht ganz so einfach ist:

`search` CycleProvider c `register` c
`where` c.serverInformation.memory > 64

Diese Regel muss nicht nur ausgewertet werden, wenn eine *CycleProvider*-Ressource registriert, aktualisiert oder gelöscht wird, sondern auch wenn die referenzierte *Server-Information*-Ressource aktualisiert wird. Wird etwa die Property `memory` der ServerInformation-Ressource von 32 auf 128 geändert, erfüllen anschließend alle CycleProvider-Ressourcen, die die geänderte Ressource referenzieren, die obige Regel.

Abbildung 4 illustriert die grundsätzliche Idee unseres Filter-Algorithmus: Sowohl Dokumente als auch Regeln werden in so genannte *Atome* zerlegt; dabei handelt es sich im Grunde um Tupel einer Tabelle. Bei Dokumenten ist ein Atom ein RDF-Statement (oder Tripel, wie in [29] beschrieben); bei Regeln besteht ein Atom aus den Regelteilen, die sich auf eine einzelne Klasse beziehen. Der Filter-Algorithmus verbindet die Atome eines Dokuments mit den Regel-Atomen, die aus der Regelbasis gewonnen wurden, und bestimmt damit alle Regeln, die möglicherweise Ressourcen des Dokuments registrieren und deshalb ausgewertet werden müssen.

Unser Publish&Subscribe-Algorithmus geht somit in drei Schritten vor: 1) Neu registrierte Dokumente müssen zerlegt werden. 2) Neu registrierte Regeln müssen zerlegt werden. 3) Dokumenten- und Regel-Atome werden verbunden und Regeln, die möglicherweise neue Ressourcen registrieren, werden inkrementell ausgewertet. Wir beschreiben jeden dieser Schritte in den folgenden Abschnitten.

Es sei betont, dass die Art und Weise, wie Regeln und Dokumente zerlegt werden nicht das Wesentliche ist. Nur das Ergebnis der Zerlegung und die gemeinsame Speicherung der zerlegten Metadaten und Regeln als relationale Daten in einem Datenbanksystem ist entscheidend. Dadurch können die Metadaten als eine Art Index auf die zerlegten Regeldaten verwendet werden, womit sich die Menge der auszuwertenden Regeln effizient feststellen lässt. Außerdem ermöglicht die Verwendung eines Datenbanksystems zur Speicherung von Metadaten und Regeln die Nutzung von Datenbank-Indexen, die einen effizienten Zugriff auf der Datenbank-Ebene unterstützen.

4.2 Die Zerlegung von Dokumenten

Jedes neu registrierte RDF-Dokument wird in seine Atome zerlegt, d. h. in RDF-Statements wie in [29] beschrieben. Zusammen mit einigen weiteren Informationen (welche für die Auswertung des Filters notwendig sind) werden sie in die Tabelle *FilterData* eingefügt (für ein Beispiel siehe Abbildung 5). Zusätzlich wird für jede Ressource ein Tupel eingefügt, das die URI-Referenz und den Klassennamen enthält (wobei das Attribut `property` auf `rdf#subject` und `value` auf die URI-Referenz gesetzt werden). Regeln können damit einzelne Ressourcen auswählen, indem sie deren URI-Referenz verwenden.

4.3 Die Zerlegung von Regeln

Eine Abonnement-Regel wird in drei Schritten zerlegt: Zuerst wird die Regel normalisiert, hauptsächlich um die Zerlegung

FilterData

uri_reference	class	property	value
doc.rdf#host	CycleProvider	rdf#subject	doc.rdf#host
doc.rdf#host	CycleProvider	serverHost	pirates.uni-passau.de
doc.rdf#host	CycleProvider	serverPort	5874
doc.rdf#host	CycleProvider	serverInformation	doc.rdf#info
doc.rdf#info	ServerInformation	rdf#subject	doc.rdf#info
doc.rdf#info	ServerInformation	memory	92
doc.rdf#info	ServerInformation	cpu	600

Abbildung 5 Tabelle *FilterData*, basierend auf dem RDF-Dokument in Abbildung 2

zu vereinfachen. Anschließend wird die normalisierte Regel in so genannte *atomare Regeln* zerlegt. Zuletzt wird ein Abhängigkeitsbaum, basierend auf den Informationen des Zerlegungsschritts, erzeugt und in einen globalen Abhängigkeitsgraphen eingefügt. Wir unterscheiden zwei Arten von atomaren Regeln: Eine *auslösende Regel* bezieht sich auf eine einzelne Klasse; sie benötigt keine Ergebnisse anderer atomarer Regeln und enthält keine Pfadausdrücke, d. h. sie enthält nur Zugriffe auf Properties. Eine *Join-Regel* repräsentiert den Join zweier Ausprägungen mit einem Join-Prädikat. Sie enthält keine weiteren Prädikate und hängt immer von zwei anderen atomaren Regeln ab.

Eine Zerlegung von (SQL-)Anfragen in ähnlich einfache Prädikate wurde bereits in System R [4] verwendet [38]. Allerdings wurden die Prädikate dort nur beim Einlesen einer Tabelle als eine Art Filter verwendet. Im MDV-System werden die Prädikate sowie alle anderen Teilregeln als relationale Daten gespeichert, um damit eine effiziente Auswertung aller Anfragen bzw. Regeln zu ermöglichen. Dies ist der entscheidende Grund, warum Regeln zerlegt werden.

Die Normalisierung ist für die Korrektheit der Regel-Zerlegung nicht unbedingt notwendig, sie vereinfacht aber ihre Erklärung und Implementierung. Wir nennen eine Regel normalisiert, wenn ihr `search`-Teil alle Klassen enthält, die in ihrem `where`-Teil benutzt werden, nicht nur direkt durch die Verwendung einer Variable, sondern auch in Pfadausdrücken. Pfadausdrücke sind in normalisierten Regeln nicht erlaubt, sondern nur Zugriffe auf Properties (einschließlich des `?`-Operators), und sie werden deshalb aufgespalten. Als Beispiel präsentieren wir die normalisierte Form der Regel aus Beispiel 1:

```
search     CycleProvider c, ServerInformation s
register   c
where      c.serverHost contains 'uni-passau.de'
           and c.serverInformation = s
           and s.memory > 64
```

4.3.1 Regelzerlegung Die Zerlegung einer Regel in atomare Regeln basiert auf ihren Prädikaten: In einem ersten Schritt werden alle Prädikate mit Konstanten aus der Original-Regel entfernt. Für jedes Prädikat wird eine auslösende Regel mit dem Prädikat als `where`-Teil erzeugt. Damit wählt die auslösende Regel alle Ressourcen aus, die das entsprechende Prädikat erfüllen. Enthält der `search`-Teil der Original-Regel Klassen ohne passendes Prädikat, wird eine auslösende Regel ohne `where`-Teil erzeugt. Anschließend wird die Original-Regel so angepasst, dass sie die Ergebnisse der auslösenden Regeln als Eingabe verwendet, anstatt die entfernten Prädikate anzuwenden. Als Beispiel diene folgende (normalisierte) Regel:

FilterRulesGT

rule_id	class	property	value
1	ServerInformation	memory	64
2	ServerInformation	cpu	500

FilterRulesCON

rule_id	class	property	value
3	CycleProvider	serverHost	uni-passau.de

Abbildung 6 Auslösende Regeln des Beispiels 4.3

```
search     CycleProvider c, ServerInformation s
register   c
where      c.serverHost contains 'uni-passau.de'
           and c.serverInformation = s
           and s.memory > 64 and s.cpu > 500
```

Für jedes Prädikate mit Konstante wird eine passende auslösende Regel erzeugt:

```
search ServerInformation s register s
where s.memory > 64                                   (RuleA)
search ServerInformation s register s
where s.cpu > 500                                     (RuleB)
search CycleProvider c register c
where c.serverHost contains 'uni-passau.de'           (RuleC)
```

Dann wird die Original-Regel angepasst:

```
search RuleA a, RuleB b, RuleC c register c
where a = b and c.serverInformation = a               (RuleD)
```

Alle verbleibenden Prädikate in der Original-Regel sind Join-Prädikate. Nacheinander wird nun jedes von ihnen entfernt und eine Join-Regel mit dem entfernen Prädikat als `where`-Teil erzeugt. Die Orginal-Regel wird wiederum angepasst. Dies geschieht solange, bis die Orginal-Regel selbst eine Join-Regel ist. In unserem Beispiel werden zwei Join-Regeln generiert:

```
search RuleA a, RuleB b register a
where a = b                                           (RuleE)
search RuleE a, RuleC c register c
where c.serverInformation = a                         (RuleF)
```

Die Abonnement-Regel ist nun in die atomaren Regeln RuleA, RuleB, RuleC, RuleE und RuleF zerlegt worden.

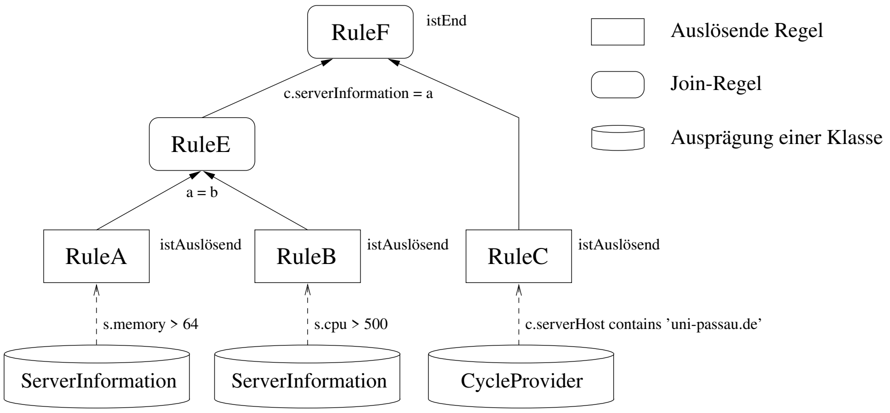

Abbildung 7 Abhängigkeitsbaum der Beispielregel aus Abschnitt 4.3

4.3.2 Erstellung eines Abhängigkeitsgraphen Die Zerlegung erzeugt azyklische Abhängigkeiten zwischen den generierten atomaren Regeln. Diese Abhängigkeiten werden in einem Abhängigkeitsbaum repräsentiert, in dem Knoten atomare Regeln und gerichtete Kanten Abhängigkeiten verkörpern. Der Baum enthält eine Endregel (eine atomare Regel, die das Ergebnis einer Abonnement-Regel liefert) als Wurzelknoten, ein oder mehrere auslösende Regeln als Blattknoten und Join-Regeln als innere Knoten. Abbildung 7 zeigt einen Abhängigkeitsbaum basierend auf den atomaren Regeln aus Abschnitt 4.3

Nach der Zerlegung werden die generierten atomaren Regeln mit allen bereits existierenden atomaren Regeln (die von früher registrierten Regeln stammen) zusammengefasst. Dies entspricht dem Zusammenfassen des Abhängigkeitsbaums der neu registrierten Regel mit dem *globalen Abhängigkeitsgraphen*, einem gerichteten, azyklischen Graphen, der aus den zusammengefassten Abhängigkeitsbäumen aller früher registrierten Regeln besteht. Durch das Zusammenfassen von Abhängigkeitsbäumen nutzt der Filter-Algorithmus Regel- und Prädikat-Redundanzen aus, d. h. er erkennt gemeinsame Teilausdrücke von Regeln und wertet diese folglich nur einmal aus.

4.3.3 Implementationsdetails Wir beschreiben nun die wichtigsten Tabellen, die vom Filter-Algorithmus verwendet werden. Wir lassen dabei Tabellen weg, die nicht direkt mit dem Algorithmus zu tun haben, etwa solche, die die Regeln speichern, die eine L-MDV registriert hat. Ein wichtiger Punkt für die effiziente Implementierung des Filters ist das physische Datenbankdesign: Zum einen werden die Filter-Tabellen als Indexe benutzt, um alle auslösenden Regeln zu bestimmen, die von neu registrierten Metadaten betroffen sind. Ausgehend von den Metadaten erlauben diese Tabellen eine effiziente Bestimmung aller auslösenden Regeln, die einen `where`-Teil haben, der bei der Auswertung mit den neuen Metadaten wahr ergibt. Zum anderen werden die Tabellen selbst mit Indexen angelegt, die einen effizienten Zugriff auf der Datenbank-Ebene unterstützen.

Die Tabelle *AtomicRules* speichert alle atomaren Regeln (siehe Abbildung 8).[3] Es existieren keine Duplikate, d. h. keine Regeln mit demselben Regeltext, aber verschiedenen rule_ids. *RuleDependencies* speichert den globalen Abhängigkeitsgraphen.

Auslösende Regeln werden außerdem in eine der Tabellen *FilterRules<u>OP</u>* bzw. *FilterRules* eingefügt, abhängig vom Operator in ihrem `where`-Teil.[4] Abbildung 6 zeigt eine Beispielausprägung der *FilterRules* und *FilterRules<u>OP</u>*-Tabellen basierend auf den auslösenden Regeln von Abschnitt 4.3.

4.4 Der Filteralgorithmus

Der Filter-Algorithmus wird gestartet, wenn ein neues Dokument registriert und zerlegt wurde. Er besteht aus zwei Schritten: Zuerst werden alle auslösenden Regeln bestimmt, die von der Registrierung neuer Metadaten betroffen sind. Anschließend werden alle Join-Regeln, die von den betroffenen auslösenden Regeln abhängen, inkrementell ausgewertet, so wie es der globale Abhängigkeitsgraph festlegt.

Die Beschreibung des Filter-Algorithmus beschränkt sich in diesem Beitrag auf das Registrieren von neuen Metadaten. Die Behandlung von Aktualisierungen und Löschungen wird in [25] ausführlich beschrieben.

Ermittlung der betroffenen auslösenden Regeln Entscheidend ist, dass eine auslösende Regel sich auf *eine* Klasse bezieht. Ihr `where`-Teil ist entweder leer oder ein Vergleich mit einer Konstante. Im ersten Fall registriert eine auslösende Regel jede neue registrierte Ressource, die eine Instanz der Klasse ist, auf die sich die Regel bezieht. Im zweiten Fall muss das Prädikat der Regel basierend auf den Atomen des Dokuments ausgewertet werden. Ein passendes Atom reicht aus, damit eine auslösende Regel betroffen ist. Unser Prototyp beginnt damit, einen Join der Tabellen *FilterData*, *FilterRules* und allen *FilterRules<u>OP</u>* zu berechnen, wobei das Join-Prädikat von der tatsächlichen *FilterRules/FilterRules<u>OP</u>*-Tabelle abhängt. Die linke Tabelle in

[3] Wir verwenden die Notation view(rule_id, class), um atomare Regeln zu referenzieren (im Gegensatz zu, z. B. RuleA).

[4] Zur Abspeicherung der Werte verschiedener Typen muß natürlich die Spalte value erweitert werden.

AtomicRules

rule_id	text
1	`search` ServerInformation s `register` s `where` s.memory > 64
2	`search` ServerInformation s `register` s `where` s.cpu > 500
3	`search` CycleProvider c `register` c `where` c.serverHost `contains` 'uni-passau.de'
4	`search` view(1, ServerInformation) a, view(2, ServerInformation) b `register` a `where` a = b
5	`search` view(4, ServerInformation) a, view(3, CycleProvider) c `register` c `where` c.serverInformation = a

RuleDependencies

source_id	target_id	param_no
1	4	1
2	4	2
3	5	2
4	5	1

Abbildung 8 *AtomicRules* und *RuleDependencies*, basierend auf Abschnitt 4.3

Initiale Iteration

uri_reference	rule_id
doc.rdf#info	1
doc.rdf#info	2
doc.rdf#host	3

Iteration 1

uri_reference	rule_id
doc.rdf#info	4

Iteration 2

uri_reference	rule_id
doc.rdf#host	5

Abbildung 9 Tabelle *ResultObjects* für eine exemplarische Ausführung des Filter-Algorithmus

Abbildung 9 zeigt das Ergebnis dieses Schritts basierend auf den vorhergehenden Beispielen. (Die Tabelle *ResultObjects* enthält stets die Ergebnisse eines Filterschritts).

Auswertung der Join-Regeln Nun werden alle Join-Regeln ausgewertet, die von betroffenen auslösenden Regeln abhängen. Join-Regeln können nicht komplett inkrementell ausgewertet werden, deshalb werden die Ergebnisse atomarer Regeln, von denen Join-Regeln abhängen, materialisiert. Die Auswertung besteht aus mehreren Iterationen. In jeder Iteration wird die Tabelle *RuleDependencies* verwendet, um atomaren Regeln zu bestimmen, die von den aktuell in der *ResultObjects* gespeicherten atomaren Regeln abhängen. Diese Regeln werden dann ausgewertet, wobei die aktuell in *ResultObjects* gespeicherten Ressourcen und materialisierte Metadaten als Eingabe verwendet werden. Das Ergebnis dieser Auswertung, d. h. Ressourcen und die atomaren Regeln, von denen sie selektiert wurden, werden wiederum in *ResultObjects* gespeichert und als Eingabe für die nächste Iteration benutzt. Ressourcen, die von einer Endregel ausgewählt werden, werden für später gesichert. Der Algorithmus terminiert, wenn keine abhängigen atomaren Regeln mehr gefunden werden. Die Terminierung ist garantiert, da der Abhängigkeitsgraph ein azyklischer, gerichteter Graph ist, so dass es einen längsten Weg von einer auslösenden Regel zu einer Endregel gibt. Die Länge dieses Pfades ist die maximale Zahl von Iterationen, die vom Filter-Algorithmus ausgeführt wird. Abbildung 9 zeigt einen exemplarischen Durchlauf des Filters, basierend auf den Tabellen aus Abschnitt 4.3.3. Der Filter terminiert mit der Ressource `doc.rdf#host` als Ergebnis. Nachdem der Filter terminiert hat, werden alle Ressourcen, die von Endregeln produziert wurden, zu den passenden L-MDVs gesendet.

5 Verwandte Arbeiten

Bereits seit mehreren Jahrzehnten werden Techniken für verteilte Datenbanksysteme untersucht; eine Übersicht dazu gibt [27]. Gemeinsam mit ObjectGlobe ist diesen Projekten die Vision, dass verteilte Systeme genauso einfach zu benutzen sein sollten wie zentrale Systeme und dass eine gute Ausführungsperformanz durch entsprechende Optimierung der Anfragen erreicht werden kann. Externe Operatoren und/oder existierende Datenquellen werden heutzutage typischerweise mit Hilfe einer Middleware-Architektur integriert. Beispiele hierfür sind Garlic [9], Information Manifold [30], TSIMMIS [35], DISCO [42] oder Tukwila [24].

Die Vorstellung eines offenen Marktplatzes, in dem verschiedene Anbieter um Anfragen konkurrieren, wurde von Mariposa [40] entliehen – auch wenn ObjectGlobe kein bestimmtes Geschäftsmodell voraussetzt. Mariposa erlaubt ebenfalls die Festlegung von QoS-Kriterien. Allerdings werden diese nur in einer Phase nach der Anfrageoptimierung berücksichtigt, nicht – wie in ObjectGlobe – in allen Phasen der Anfragebearbeitung. Weitere Informationen zu QoS in ObjectGlobe sind in [6] zu finden.

Metadaten-Repositories werden seit vielen Jahren von Datenbanksystemen eingesetzt, um Metadaten über Tabellen, Indexen, usw. zu speichern [45]. Das Internet forcierte die Entwicklung neuartiger Metadaten-Verwaltungs-Systeme. Ein Beispiel hierfür ist UDDI [43], ein System zur Verwaltung von Metadaten über Web-Services. Im Gegensatz zum MDV-System verwendet es allerdings ein festes Schema und bietet keine Benachrichtigung bei der Änderung von Daten. WebSemantics [33] durchsucht das Internet nach HTML-Dateien, die Metainformationen über Datenquellen enthalten, und fasst sie zu Katalogen zusammen. Das Middleware-System MOCHA [37] nutzt ein (zentrales) Repository, um Code in Form von Java-Klassen und die dazugehörige Dokumentation in `RDF` zu verwalten. Der Secure Service Discovery Service [12] speichert Metadaten über Netzwerkdienste im `XML`-Format. Dienste wie JINI [3], UPnP [44] und SLP [21] erlauben die Suche nach Plug-And-Play-Diensten.

Unser Filter-Algorithmus verwendet so genannte auslösende Regeln als Index auf alle Abonnement-Regeln, die sich auf geänderte Metadaten beziehen. Le Subscribe [36, 15] verwendet einen ähnlichen Ansatz, allerdings im Rahmen eines Hauptspeicheralgorithmus. Das Gryphon-System [1] baut aus Anfragen einen Baum auf, in dem jeder Knoten einem Vergleich entspricht. [22] präsentiert einen Interval Binary Search Tree zur effizienten Verwaltung von Intervallen. Ähnlich wie das MDV-System baut auf SIFT [46] nicht auf einem festen Datenschema auf. Es erlaubt das Verbreiten von beliebigen Textdokumenten. [2] präsentiert einen Filter-Algorithmus, der XQL-Anfragen verwendet, um bestimmte XML-Dokumente aus einem Strom von Dokumenten auszuwählen. Unserem Wissen nach erlaubt keines dieser Systeme Referenzen zwischen den Informationen, die verbreitet werden, d. h. zwischen verschiedenen Dokumenten, wie es das MDV-System tut. [41] beschreibt die kontinuierliche Auswertung von Anfragen auf einer Datenbank, eingeschränkt auf Datenbanken, in die nur eingefügt wird. NiagaraCQ [10] und OpenCQ [32] erweiterten diesen Ansatz, so dass auch das Aktualisieren und Löschen von Daten möglich ist, ähnlich wie im MDV-System. Der Cache eines MDV-Systems kann als Menge von materialisierten Sichten betrachtet werden. Algorithmen für die Aktualisierung solcher Sichten werden in [20,5,31] beschrieben. Es existieren auch Ähnlichkeiten mit Semantic Caching [13].

6 Zusammenfassung und Ausblick

In dieser Arbeit haben wir das ObjectGlobe-System präsentiert, ein offenes und verteiltes Anfragebearbeitungssystem auf Internet-Datenquellen. Unser System erweitert die begrenzten Fähigkeiten des Internets bezüglich Anfragebearbeitung, indem es einen offenen Marktplatz schafft, in dem eine Vielzahl von Anbietern Rechenzeit, Daten und Anfrageoperatoren bereitstellen.

Den Schwerpunkt dieses Beitrags hat die Metadatenverwaltung MDV gebildet, ein verteiltes Metadaten-Management-System. MDV besitzt eine 3-schichtige Architektur und unterstützt Caching und Replikation in der Mittelschicht, so dass Informationen nahe bei den Benutzern gespeichert und Anfragen lokal ausgewertet werden können. Das Hinzufügen oder Entfernen von Servern aus der Mittelschicht ermöglicht die leichte Anpassung unseres Systems an wechselnde Lastsituationen. Um Replikate aktuell zu halten und die Replikation von neuen und relevanten Informationen zu initiieren, verwendet das MDV-System einen neuen Publish&Subscribe-Algorithmus. Wir haben diesen Algorithmus im Detail beschrieben und gezeigt, wie er mit Hilfe eines normalen relationalen Datenbanksystems implementiert werden kann.

Sowohl das ObjectGlobe- als auch das MDV-System wurden inzwischen fertig gestellt und als Kernkomponenten in das ServiceGlobe-Projekt übernommen. Das ServiceGlobe-Projekt zielt darauf ab, eine Plattform für die Entwicklung und Ausführung von E-Services zu entwickeln. E-Services sind im Prinzip einfache Software-Komponenten, die gleichartige Funktionalität zusammenfassen und auf die über Standard-Internet-Protokolle zugegriffen werden kann. ServiceGlobe ermöglicht die Komposition von neuen, zusammengesetzten E-Services, basierend sowohl auf bereits existierenden als auch auf (einfacheren) zusammengesetzten E-Services. Die Komposition kann mit entsprechenden Sprachen (z. B. XL [17]) oder einer Graph-orientierten Notation erfolgen. ServiceGlobe ermöglicht die verteilte Ausführung von E-Services innerhalb des ServiceGlobe-Netzes, d. h. auf allen Rechnern, auf denen ein ServiceGlobe-Server läuft. Zusammengesetzte E-Services müssen als Ziel eines Aufrufes keinen tatsächlichen E-Service angeben, es reicht die Festlegung einer technischen Spezifikation, der der aufzurufende E-Service entsprechen soll. Mit Hilfe von UDDI [43] werden dann bei der Ausführung ein oder mehrere passende E-Services ausgewählt und aufgerufen, wobei QoS-Kriterien die Auswahl der E-Services zusätzlich beeinflussen können.

Literatur

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 53–61, 1999.
2. M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 53–64, Cairo, Egypt, September 2000.
3. K. Arnold. *The Jini(TM) Specification (The Jini(TM) Technology Series)*. Addison-Wesley, Reading, MA, USA, 1999.
4. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Trans. on Database Systems*, 1(2):97–137, June 1976.
5. J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–71, Washington, D.C., USA, 1986.
6. R. Braumandl. *Quality of Service and Query Processing in an Information Economy*. PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, D-94030 Passau, 2001. Universität Passau.
7. R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal: Special Issue on E-Services*, 10(3):48–71, August 2001.
8. D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Candidate Recommendation `http://www.w3.org`, WWW-Consortium, 2000.
9. M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, pages 124–131, March 1995.
10. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 379–390, Dallas, USA, June 2000.
11. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl/`, March 2001.
12. S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. of ACM MOBICOM Conference*, pages 24–35, Seattle, USA, August 1999.

13. S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.
14. T. Dierks and C. Allen. The TLS Protocol Version 1.0. `ftp://ftp.isi.edu/in-notes/rfc2246.txt`, January 1999.
15. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 115–126, Santa Barbara, USA, May 2001.
16. D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engeneering Bulletin*, 22(3):27–34, September 1999.
17. D. Florescu and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. *IEEE Data Engineering Bulletin*, 24(2):48–56, 2001.
18. A. Frier, P. Karlton, and P. Kocher. *The SSL 3.0 Protocol*. Netscape Communications Corp., `http://home.netscape.com/eng/ssl3`, November 1996.
19. G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
20. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 157–166, Washington, D.C., USA, May 1993.
21. E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. `http://www.rfc-editor.org/rfc/rfc2608.txt`, June 1999.
22. E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–280, May 1990.
23. R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. `http://www.rfc-editor.org/rfc/rfc2459.txt`, January 1999.
24. Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An Adaptive Query Execution Engine for Data Integration. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 299–310, Philadelphia, PA, USA, June 1999.
25. M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proc. of the 18th Intl. Conference on Data Engineering (ICDE)*, pages 309–320, 2002.
26. A. Kemper and C. Wiesner. HyperQueries: Enabling Flexible Distributed Query Processing on the Internet. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 551–560, 2001.
27. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.
28. D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1):43–82, March 2000.
29. O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation `http://www.w3.org`, WWW-Consortium, February 1999.
30. A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 251–262, Bombay, India, September 1996.
31. B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A Snapshot Differential Refresh Algorithm. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 53–60, Washington, USA, June 1986.
32. L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.
33. G. A. Mihaila, L. Raschid, and A. Tomasic. Equal Time for Data on the Internet with WebSemantics. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science (LNCS)*, pages 87–101, Valencia, Spain, March 1998. Springer-Verlag.
34. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1999.
35. Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 161–186, December 1995.
36. J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Proc. of the IFCIS International Conference on Cooperative Information Systems*, Eilat, Israel, September 2000.
37. M. Rodriguez-Martinez and N. Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 213–224, Dallas, USA, June 2000.
38. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.
39. S. Seltzsam, S. Börzsönyi, and A. Kemper. Security for Distributed E-Service Composition. In *Proc. of the 2nd Intl. Workshop on Technologies for E-Services (TES)*, volume 2193 of *Lecture Notes in Computer Science (LNCS)*, pages 147–162, Rome, Italy, 2001. Springer.
40. M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *The VLDB Journal*, 5(1):48–63, January 1996.
41. D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 321–330, 1992.
42. A. Tomasic, L. Raschid, and P. Valduriez. Scaling Acccess to Distributed Heterogeneous Data Sources with DISCO. *IEEE Trans. Knowledge and Data Engineering*, 10(5):808–823, October 1998.
43. Universal Description, Discovery and Integration (UDDI) Technical White Paper. White Paper, Ariba Inc., IBM Corp., and Microsoft Corp., September 2000. `http://www.uddi.org/`.
44. Universal Plug and Play Device Architecture. Document, Microsoft Corporation, June 2000. `http://www.upnp.org`.
45. R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R^*: An Overview of the Architecture. IBM Research, San Jose, CA, RJ3325, December 1981. Reprinted in: M. Stonebraker (ed.), Readings in Database Systems, Morgan Kaufmann Publishers, 1994, pp. 515–536.
46. T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Trans. on Database Systems*, 24(4):529–565, 1999.