Verteilte Metadatenverwaltung für die Anfragebearbeitung auf Internet-Datenquellen

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

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

Zusammenfassung Durch die Zunahme von E-Commerce-Anwendungen verstärkt sich die Bedeutung der verteilten Datenverarbeitung von Datenquellen im Internet. Viele Firmen schließen sich zu virtuellen Unternehmen zusammen oder gründen gemeinsam virtuelle Marktplätze. Dabei erschweren ständige Veränderungen der verfügbaren Ressourcen (z. B. Datenquellen) und ein permanent fluktuierender Teilnehmerkreis das Auffinden relevanter Informationen. In dieser Arbeit präsentieren wir die Metadatenverwaltung MDV, deren Publish/Subscribe-Mechanismus die Dynamik im Bereich E-Commerce adressiert. Ein Backbone mit wenigen öffentlichen MDVs übernimmt die Registrierung und Verteilung von Metadaten im `RDF`-Format. Lokale MDVs abonnieren Metadaten von einem Backbone. Wir zeigen anhand des offenen, verteilten Anfragebearbeitungssystems ObjectGlobe, wie die MDV das Auffinden wichtiger Ressourcen für die Anfragebearbeitung auf Internet-Datenquellen ermöglicht. Eine weitere zentrale Herausforderung in der Entwicklung unternehmensübergreifender Datenverarbeitungsanwendungen ist Sicherheit, insbesondere Authentifizierung und Autorisierung. Aus diesem Grund gehen wir näher auf den Einsatz der MDV zur Speicherung von Sicherheitsinformationen in ObjectGlobe ein.

1 Einleitung

Durch die Zunahme von E-Commerce-Anwendungen verstärkt sich die Bedeutung der verteilten Datenverarbeitung von Datenquellen im Internet. Die Dynamik in diesem Bereich veranlasst viele Firmen dazu, sich zu virtuellen Unternehmen zusammenzuschließen oder gemeinsam virtuelle Marktplätze zu gründen. Dabei erschweren ständige Veränderungen der verfügbaren Ressourcen (z. B. Datenquellen) und ein permanent fluktuierender Teilnehmerkreis das Auffinden relevanter Informationen. Inkonsistenzen und die schnelle Alterung der Daten tragen zusätzlich zu diesem Problem bei.

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. Die Art der Anfragebearbeitung, wie sie ObjectGlobe anbietet, ist eine unverzichtbare Technik für skalierbare Internetanwendungen wie etwa Business-to-Business (B2B) E-Commerce-Systeme. Ein Beispiel für ein solches System ist der virtuelle Marktplatz mySAP.com [SAP99].

Wir haben die Metadatenverwaltung MDV als eine Basistechnologie für Systeme wie ObjectGlobe oder E-Commerce-Anwendungen entwickelt. Sie wird in ObjectGlobe hauptsächlich zur Verwaltung von Ressourcen für die Internet-Anfragebearbeitung verwendet. Trotz ihrer verteilten Architektur wertet die MDV Anfragen ausreichend schnell aus, um während des Parsens und Optimierens von Anfragen eingesetzt werden zu können. Die MDV besteht aus wenigen öffentlichen Servern, genannt *öffentliche MDVs*, und einer beliebigen Zahl von lokalen Servern, *lokale MDVs* genannt. Letztere ermöglichen durch Caching den effizienten Zugriff auf gespeicherte Metadaten. Eine Publish/Subscribe-Komponente erlaubt lokalen MDVs die explizite Auswahl der zu puffernden Metadaten. Dazu abonnieren sie Metadaten von öffentlichen MDVs mit Hilfe einer Regelsprache. Die Publish/Subscribe-Komponente sorgt außerdem für die Erhaltung der Cache-Konsistenz. Öffentliche MDVs verteilen Änderungen an ihren Metadaten automatisch an alle lokalen MDVs, basierend auf den Abonnement-Regeln. Da die Zahl der Regeln sehr groß werden kann und — theoretisch — alle ausgewertet werden müssten, entwickelten wir einen Vorfilter-Algorithmus. Basierend auf den geänderten Metadaten bestimmt dieser eine Obermenge der Abonnement-Regeln, die ausgewertet werden müssen. Dadurch wird eine effiziente Verteilung der geänderten Metadaten erreicht. Wir verwenden `RDF` (Resource Description Framework) als Metadatenmodell und die empfohlene `XML`-Syntax zur Speicherung von `RDF`-Dokumenten [LS99]. `RDF` wurde vom W3C entwickelt, um eine standardisierte Beschreibung von Metadaten aller Art – vor allem im Internet – zu ermöglichen.

Auch andere Metadaten-Managementsysteme verwenden `RDF` und `XML` zur Speicherung von Metadaten, etwa das Middleware-System MOCHA [RMR00]. Dieses nutzt eine (zentrale) Metadatenverwaltung, um Code in Form von Java-Klassen und die dazugehörige Dokumentation in `RDF` zu verwalten. UDDI (Universal Description, Discovery and Integration) [UDD00], ein kommerzielles Projekt mehrerer Firmen, verwaltet Informationen über allgemeine, im Internet zur Verfügung stehende Dienste im XML-Format; es stellt allerdings keinen Mechanismus zur automatischen Propagierung von Änderungen zur Verfügung. WebSemantics [MRT98] ist eine Metadatenverwaltung, die das Internet nach HTML-Dateien durchsucht, die Metainformationen über Datenquellen enthalten, und sie zu Katalogen zusammenfasst. Der Secure Service Discovery Service [CZH$^+$99] speichert Metadaten über Netzwerkdienste im `XML`-Format. Ähnliche Service-Discovery-Dienste sind JINI [Arn99], UPnP [UPN00] und SLP [GPVD99]. Die Idee, im Vorfilter Teile von Regeln zusammenzufassen, stammt aus NiagaraCQ [CDTW00]. Dieses System ermöglicht, wie auch [LPT99, TGNO92], die kontinuierliche Auswertung von Anfragen auf einer Datenbank. Der Vorfilter der MDV entspricht den Matching-Algorithmen, die in Publish/Subscribe-Systemen verwendet werden [PFLS00, ASS+99, YGM99, AF00]. Allerdings erlaubt er die Verwendung von Referenzen zwischen den Daten, die verteilt werden. Dies ermöglicht unseres Wissens nach kein anderer Algrithmus.

Sicherheit spielt im Internet eine zunehmend wichtigere Rolle. Für E-Commerce-Anwendungen und virtuelle Marktplätze ist ein effizientes Sicherheitssystem eine Notwendigkeit, um Daten und Privatsphäre der Benutzer zu schützen. Wir beschreiben in dieser Arbeit deshalb den Einsatz der MDV zur Speicherung von Informationen im Rahmen des Sicherheitssystems von ObjectGlobe und zur Verteilung von lokalen Sicherheitsinformationen, um sie bei der Anfragebearbeitung auf Internet-Datenquellen zu verwenden.

In föderierten DBMSen [SL90, CSS99], die Ähnlichkeiten mit ObjectGlobe aufweisen, unterscheidet man zwischen dem globalen Sicherheitssystem der Föderation und den lokalen Sicherheitssystemen der einzelnen Komponenten-DBMSen. Auf globaler Ebene werden häufig Views eingesetzt, um Rechte für globale Datenobjekte anzugeben [SL90, TLW87, WS87]. Dort wie auch in [WL93, Per92] und [JD95, JD93, SST+99] werden (globale) Anfragen zuerst auf dieser Ebene überprüft. Nach Zerlegung der globalen Anfrage werden die einzelnen lokalen Teilanfragen nochmals von den jeweiligen lokalen Sicherheitssystemen überprüft. Der zweite Verifikationsvorgang kann, abhängig von der Art des DBMSs, entfallen. Die Anforderungen in unserem verteilten und offenen Anfragebearbeitungssystem ObjectGlobe gehen über die Problemstellung in föderierten DBMSen allerdings hinaus. Beispielsweise ist der Kreis der Anbieter ständigen Veränderungen unterworfen, d. h. immer wieder kommen neue Anbieter (etwa von Datenquellen) hinzu oder Alte fallen weg. Außerden besitzen die verschiedenen Anbieter keinen gemeinsamen Benutzerkreis und keine gemeinsame Sicherheitspolitik, es existiert also keine zentrale Verwaltung.

Der Rest dieser Arbeit ist wie folgt gegliedert: Abschnitt 2 präsentiert einen Überblick über das ObjectGlobe-System. Abschnitt 3 erläutert die Metadatenverwaltung MDV, ihren Publish/Subscribe-Mechanismus und ihren Vorfilteralgorithmus. Abschnitt 4 beschreibt das Sicherheitssystem von ObjectGlobe und die Verteilung und Integration von Sicherheitsanforderungen durch die MDV. Abschnitt 5 beschließt diese Arbeit mit einer Zusammenfassung.

2 Das ObjectGlobe-System für verteilte Anfragebearbeitung

In dieser Arbeit verwenden wir ObjectGlobe, ein offenes, verteiltes Anfragebearbeitungssystem für Internet-Datenquellen, als einen möglichen Klienten der MDV, anhand dessen wir die einzelnen Komponenten beschreiben. Das Ziel beim Entwurf von ObjectGlobe war die Entwicklung eines Systems, mit dem im Internet jede Art von Anfrageoperator auf jedem Rechner ausgeführt und auf jede Art von Daten angewendet werden kann. Die zugrunde liegende Idee ist die Schaffung eines offenen Marktes für drei Arten von Anbietern, auch *Provider* genannt: *Data-Provider* bieten Daten an, *Function-Provider* stellen Anfrageoperatoren zur Verfügung, mit denen Daten verarbeitet werden können, und *Cycle-Provider* stel-

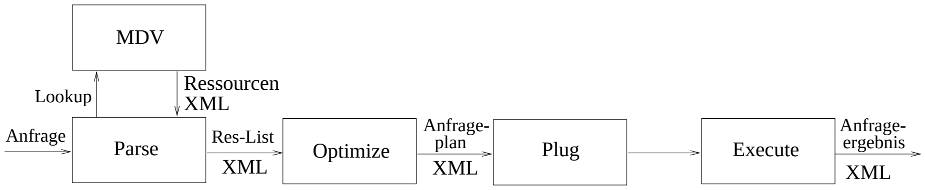

Abbildung 1. Ausführen einer Anfrage in ObjectGlobe

len Rechenzeit zur Ausführung von Anfragen zur Verfügung. Es ist zwar möglich, dass ein einzelner Rechner alle drei Dienste anbietet, für den Rest dieser Arbeit werden die Provider aber als getrennte Einheiten betrachtet. Eine ausführliche Beschreibung des ObjectGlobe-Systems ist in [BKK+99a] zu finden.

2.1 Anfragebearbeitung in ObjectGlobe

Die Auswertung einer Anfrage erfordert in ObjectGlobe vier Schritte (Abbildung 1):

1. Lookup: In dieser Phase werden von der MDV Metadaten über alle Datenquellen, Cycle-Provider und Operatoren angefordert, die bei der Ausführung der Anfrage nützlich sein könnten. Zusätzlich stellt die MDV Sicherheitsanforderungen zur Verfügung (siehe Abschnitt 4). 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 dem Iteratormodell [Gra93] 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 [BKK+99a].

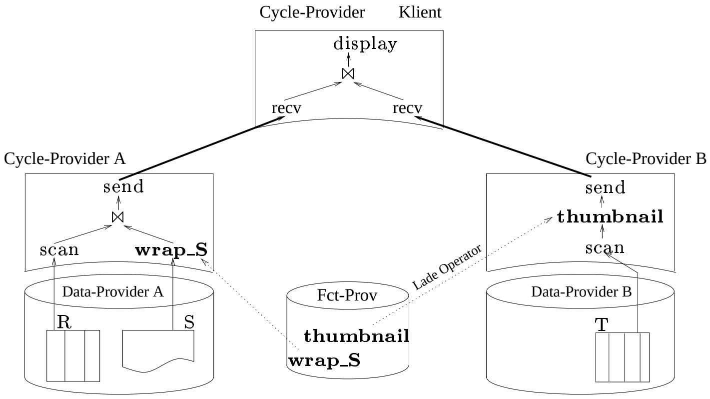

Abbildung 2. Verteilte Anfragebearbeitung in ObjectGlobe

2.2 Ein Beispiel

Zur Illustration der Anfragebearbeitung in ObjectGlobe dient das Beispiel aus Abbildung 2. Es zeigt zwei Data-Provider, A und B, und einen Function-Provider. Wir nehmen an, dass die Data-Provider zugleich Cycle-Provider sind, sodass 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 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.

2.3 Der ObjectGlobe-Optimierer

Die Aufgabe des Optimierers ist die Erzeugung eines effizienten Anfrageplans unter Berücksichtigung von Informationen darüber, welche Rechte ein Benutzer bei den Providern besitzt. Im Folgenden bezeichnen wir diese Informationen als *Sicherheitsanforderungen*. Normalerweise wird vor der Optimierung überprüft, ob ein Benutzer eine bestimmte Anfrage ausführen darf. Zu diesem Zeitpunkt sind aber weder die tatsächlichen Anfrageoperatoren bekannt noch die Cycle-Provider, die in der Anfrage benutzt werden. Da somit vor der Optimierung keine Autorisierung möglich ist, muss der Optimierer während der Optimierung neben anderen Randbedingungen (z. B. Antwortzeit oder Quality-of-Service-Parameter) auch Sicherheitsanforderungen berücksichtigen.

Einige Randbedingungen werden vom Benutzer vorgegeben (Quality-of-Service-Parameter) und dem Optimierer zusammen mit der Anfrage übergeben. Andere Randbedingungen, etwa die Ressourcen einzelner Provider (Größe des Haupt- und Hintergrundspeichers), werden bei der Registrierung des Providers

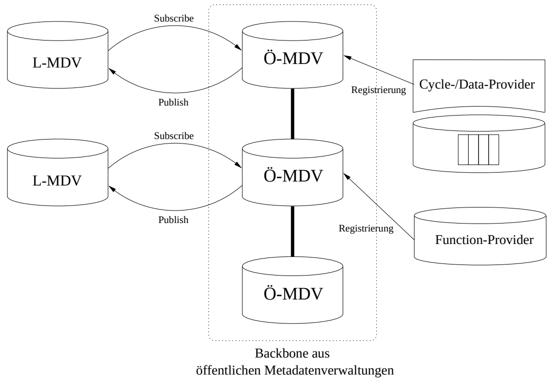

Abbildung 3. Die Architektur der Metadatenverwaltung MDV

in der MDV angegeben. Die MDV enthält außerdem Informationen über verfügbare Relationen, Anfrageoperatoren, Cycle-Provider usw. Diese Daten werden vor dem Optimierungsvorgang bei der MDV angefordert und zur Optimierung verwendet.

Das Sicherheitssystem von ObjectGlobe verwendet lokale Autorisierung, d. h. ein Provider entscheidet autonom, ob er die lokale Ausführung einer Anfrage erlaubt. Die Sicherheitsanforderungen liegen also lokal bei einem Provider. ObjectGlobe benutzt die Metadatenverwaltung MDV, um Sicherheitsanforderungen für die Anfrageoptimierung zugänglich zu machen (siehe Abschnitt 4).

3 Die Metadatenverwaltung

Die Metadatenverwaltung MDV bietet einen verteilten Dienst zur Registrierung, Verwaltung und Abfrage von Metadaten an. Die MDV verwendet `RDF` zur Spezifikation sowohl des *Schemas der Metadaten* als auch der *Metadaten selbst*. Mit RDF Schema [BG99] wird das Metadaten-Schema definiert, das aus einer Menge von Klassen und ihren Attributen (in `RDF` Properties genannt) besteht. Jede MDV besitzt genau ein Metadaten-Schema, mit dem Metadaten konform sein müssen, wenn man sie registrieren will. Das ObjectGlobe-Schema [BKK+00] definiert beispielsweise die Klassen `CycleProvider`, `DataProvider`, `Partition`, `FunctionProvider`, usw. Die Metadaten selbst werden durch `RDF`-Dokumente repräsentiert. Jedes `RDF`-Dokument definiert eine Menge von Objekten (in `RDF` auch Ressourcen genannt), die Instanzen der Klassen des Metadaten-Schemas sind. Jedes Objekt besitzt eine innerhalb des Dokuments eindeutige ID. Zusammen mit der URL des `RDF`-Dokuments ist diese ID global eindeutig.

Die Skalierbarkeit des Systems garantiert eine zweistufige Architektur. Abbildung 3 zeigt neben dieser auch einen Cycle-/Data- und einen Function-Provider des ObjectGlobe-Systems, die gerade Metadaten registrieren.

Öffentliche Metadatenverwaltungen: Das Backbone aus öffentlichen Metadatenverwaltungen (Ö-MDVs) speichert globale Metadaten, die im Internet frei zugänglich sind. Die Metadaten der Backbone-MDVs werden stets konsistent gehalten, d. h. sie sind auf den Ö-MDVs des Backbones repliziert. Änderungen an einer Ö-MDV werden an die anderen Ö-MDVs des Backbones propagiert.

Lokale Metadatenverwaltungen: Eine lokale Metadatenverwaltung (L-MDV) *abonniert* einen Teil der Metadaten eines Backbones. Die Auswahl der Metadaten erfolgt mit Hilfe einer Menge von Regeln, die die L-MDV bei einer Ö-MDV registriert (*Subscribe*-Schritt). Die abonnierten Daten werden in einem lokalen Puffer gespeichert[1]. Anhand der registrierten Regeln erkennt eine Ö-MDV, welche L-MDVs von neu registrierten, geänderten oder gelöschten Metadaten benachrichtigt werden müssen (*Publish*-Schritt).
Zusätzlich speichert eine L-MDV auch lokale Metadaten, die nicht einer breiten Öffentlichkeit zugänglich sein sollen. Diese Metadaten werden *nicht* an das Backbone weitergeleitet.

Die MDV stellt eine (deklarative) Anfragesprache[2] zur Verfügung. Anfragen in dieser Sprache werden *Lookup-Anfragen* genannt und von L-MDVs ausgewertet, die dabei sowohl lokale als auch gepufferte Metadaten verwenden.

Die MDV ist in Java implementiert und setzt auf ein relationales DBMS auf. In einer Initialisierungsphase wird das Metadaten-Schema in ein relationales Schema transformiert. Beim Einfügen werden die Metadaten eines `RDF`-Dokuments aufgeteilt und als Tupel in die verschiedenen Relationen eingefügt. Lookup-Anfragen werden von der MDV in SQL-Joinanfragen über mehrere Relationen umgewandelt.

3.1 Registrieren, Ändern und Löschen von Metadaten

Die Registrierung von Metadaten in Form eines `RDF`-Dokuments besteht aus drei Schritten:

1. Die Ö-MDV, bei der registriert wird, prüft die neuen Metadaten auf ihre syntaktische Korrektheit und Konsistenz. Anschließend fügt sie die Daten in ihre Datenbank ein und benachrichtigt alle weiteren Ö-MDVs des Backbones von der Registrierung.
2. Im nächsten Schritt bestimmt die Ö-MDV alle L-MDVs, die sie basierend auf der registrierten Regelmenge von der Metadaten-Registrierung benachrichtigen muss. Das sind alle L-MDVs, bei denen die Auswertung ihrer Regelmenge mit den neuen Metadaten ein nicht-leeres Ergebnis liefert. Zur effizienten Ermittlung der Regeln, die von der Registrierung neuer Metadaten betroffen sind, wird ein Vorfilteralgorithmus eingesetzt. Aus diesen Regeln können anschließend die betroffenen L-MDVs bestimmt werden.

[1] Der Puffer einer L-MDV kann prinzipiell als Menge von materialisierten Sichten [GMS93, BLT86, LHM+86, Han87] betrachtet werden (mit den Regeln als Sicht-Definitionen).

[2] Die Anfragesprache wird aus Platzgründen nicht vorgestellt. Sie ähnelt allerdings stark der Regelsprache, die in Abschnitt 3.2 erläutert wird.

3. Im letzten Schritt werden die neuen Daten an die ermittelten L-MDVs übertragen.

Das Ändern oder Löschen von Metadaten wird analog durchgeführt. Dabei können immer nur komplette `RDF`-Dokumente modifiziert oder gelöscht werden.

3.2 Das Regelsystem

Mit einer Regelmenge legt eine L-MDV fest, welche Metadaten abonniert und lokal gepuffert werden. Der Administrator einer L-MDV gibt eine Grundmenge von Regeln entsprechend den lokalen Bedürfnissen vor; Benutzer der L-MDV können außerdem gezielt Ressourcen (z. B. eine bestimmte Datenquelle oder Anfrageoperatoren eines speziellen Function-Providers im Falle von ObjectGlobe) auswählen, die zusätzlich gepuffert werden sollen. Die L-MDV überträgt die Regelmenge an die Ö-MDV, die sie auswertet und die Ergebnisse an die L-MDV zurückschickt. Regeln haben die folgende Syntax:

$$\begin{array}{l}\texttt{search}\ Class_1\ c_1,\ Class_2\ c_2,\ \ldots,\ Class_n\ c_n\ \texttt{register}\ c_i \\ \texttt{where}\ \ Bedingung(c_1,\ c_2,\ \ldots,\ c_n)\end{array}$$

Eine L-MDV legt damit fest, dass sie alle Objekte der Klasse $Class_i$ puffern will, die die Bedingung $Bedingung(c_1,\ c_2,\ \ldots,\ c_n)$ erfüllen. Die Bedingung besteht aus einem Bool'schen Ausdruck mit Objekten der Klassen $Class_1$ bis $Class_n$ und den Operatoren `AND`, `()`, `=`, `<`, `>` und `contains` (prüft, ob der String auf der linken Seite den String der rechten Seite enthält). Die derzeitige Implementierung unterstützt kein `OR`, man kann allerdings eine Regel mit `OR`-Operator mit Hilfe der Bool'schen Algebra leicht in mehrere Regeln aufteilen, die kein `OR` enthalten.

Als Beispiel diene folgende Regel, mit der eine L-MDV alle ObjectGlobe-Datenquellen (Partitionen) abonniert, die das Thema *Hotels* haben und mehr als 1000 Tupel enthalten:

```
search Partition p register p
where  ( p.theme.themeName = 'Hotels' AND p.cardinality > '1000' )
```

3.3 Der Vorfilteralgorithmus

Die Zahl der registrierten Regeln in einer Ö-MDV kann sehr groß werden, da zum einen sehr viele L-MDVs Metadaten einer Ö-MDV abonnieren und zum anderen schon die Regelmenge einer einzelnen L-MDV sehr umfangreich werden kann. Wenn Metadaten registriert, geändert oder gelöscht werden, müssen theoretisch alle registrierten Regeln ausgewertet werden. Da dies zu aufwendig ist, verwendet die MDV einen Vorfilteralgorithmus, der die Menge der auszuwertenden Regeln einschränkt. Außerdem erfolgt die Auswertung der Regeln inkrementell, d. h. sie werden – soweit möglich – nur unter Verwendung der neuen, geänderten oder zu löschenden Daten ausgewertet. Im Folgenden werden die einzelnen Schritte des Vorfilteralgorithmus näher beschrieben.

Zerlegung der Regeln Jede registrierte Regel wird in *atomare Teilregeln* zerlegt. Ursprünglich stammt die Idee, Teile von Anfragen zusammenzufassen und gemeinsam auszuwerten, aus [CDTW00]. Dort werden identische Teilpläne mehrerer XML-QL-Anfrageplänen zu Gruppen zusammengefasst und gemeinsam ausgewertet. Der Ansatz der MDV unterscheidet sich durch die Verwendung einer eigenen Regelsprache allerdings stark von dem in [CDTW00] beschriebenen, sodass nur noch die ursprüngliche Idee erhalten geblieben ist, Teilausdrücke von Anfragen zusammenzufassen. Wir unterscheiden drei Arten atomarer Teilregeln:

Auslösende Teilregeln: Eine derartige Teilregel bezieht sich ausschließlich auf eine Klasse, benötigt keine Ergebnisse anderer Teilregeln und verwendet keine Pfadausdrücke. Die Teilregel enthält also nur Attributzugriffe und möglicherweise den `?`-Operator. Der `?`-Operator kann nur auf mengenwertige Attribute angewendet werden und repräsentiert ein beliebiges Element des mengenwertigen Attributs[3]. Ein Beispiel für auslösende atomare Teilregeln sind RegelB, RegelD und RegelF aus Beispiel 1 (siehe weiter unten).

Abhängige Teilregeln: Wie eine auslösende Teilregel bezieht sich eine abhängige Teilregel ausschließlich auf eine Klasse und enthält keine Pfadausdrücke. Allerdings benötigt sie das Ergebnis mindestens einer anderen Teilregel. Ein Beispiel für eine abhängige atomare Teilregel ist RegelG aus Beispiel 1.

Jointeilregeln: Eine Jointeilregel beinhaltet einen Join zwischen zwei Klassen mit einem Joinprädikat. Die Teilregel darf keine weiteren Prädikate (etwa Selektionen) enthalten. RegelC aus Beispiel 1 ist eine atomare Jointeilregel.

Bei der Zerlegung von Regeln werden einzelne Teilregeln benannt und können dann zusammen mit Klassen im `search`-Teil von anderen Teilregeln verwendet werden. In diesem Fall wird die entsprechende Variable nicht an *alle* Objekte einer Klasse gebunden, sondern nur an diejenigen, die sich als Ergebnis der Teilregel ergeben. Jede Regel kann nach folgendem Schema schrittweise in atomare Teilregeln zerlegt werden (Die Zerlegungsschritte sind nach absteigender Priorität geordnet):

1. **Regeln ohne Bedingung:**
   `search` *Class* c `register` c
   Eine derartige Regel ist bereits eine auslösende atomare Teilregel, d. h. sie ist nicht weiter zerlegbar.
2. **Regeln mit einer Konjunktion einfacher Vergleiche:**
   `search` *Class* c `register` c
   `where` c.x ∘ `String` `AND` c.y ∘ `String` ...
   (mit $\circ \in \{$`=`, `<`, `>`, `contains`$\}$). Derartige Regeln sind ebenfalls auslösende atomare Teilregeln.
3. **Regeln mit `?` in einem Pfadausdruck:**
   `search` *Class* c `register` c `where` c.x.?.Path ∘ `String`
   Regeln dieser Art werden zerlegt in eine Teilregel der Form

[3] Mengenwertige Attribute in der MDV sind typisiert, d. h. alle Elemente der Menge müssen denselben Typ besitzen. In `RDF` ist dies nicht zwingend erforderlich.

RegelA: `search` *Class(c.x.?)* d `register` d `where` d.Path $\circ$ `String`

und die abhängige atomare Teilregel

`search` *Class* c, RegelA a `register` c `where` c.x.? = a

Class(c.x.?) bezeichnet hier die Klasse der Elemente des mengenwertigen Attributs c.x.

4. **Regeln mit Pfadausdrücken ohne ?:**

   `search` *Class* c `register` c `where` c.x.Path $\circ$ `String`

   Eine solche Regel wird zerlegt in die Teilregel

   RegelA: `search` *Class(c.x)* d `register` d `where` d.Path $\circ$ `String`

   und die abhängige atomare Teilregel

   `search` *Class* c, RegelA a `register` c `where` c.x = a

5. **Regeln mit einer Konjunktion zweier Bedingungen:**

   `search` *Class* c `register` c `where` $Bedingung_1(c)$ `AND` $Bedingung_2(c)$

   Eine solche Regel wird zerlegt in die Teilregeln

   RegelA: `search` *Class* c `register` c `where` $Bedingung_1(c)$
   RegelB: `search` *Class* c `register` c `where` $Bedingung_2(c)$

   und die abhängige atomare Joinregel

   `search` RegelA a, RegelB b `register` a `where` a = b

   Regeln mit mehr als zwei Bedingungen können durch eine entsprechende Klammerung auf diesen Fall zurückgeführt werden.

6. **Joinregeln:**

   `search` $Class_1$ c, $Class_2$ d `register` c
   `where` $Bedingung_1(c)$ `AND` $Bedingung_2(d)$ `AND` $Joinbedingung(c, d)$

   Regeln dieser Art werden zerlegt in Teilregeln der Form

   RegelA: `search` $Class_1$ c `register` c `where` $Bedingung_1(c)$
   RegelB: `search` $Class_2$ d `register` d `where` $Bedingung_2(d)$

   und die Joinregel

   `search` RegelA a, RegelB b `register` a `where` $Joinbedingung(a, b)$

   Ist eine Bedingung leer, wird die entsprechende Teilregel ohne `where`-Teil erzeugt. Im Falle einer leeren Joinbedingung wird also ein Kreuzprodukt berechnet. Joins über mehrere Klassen können analog den Zerlegungsregeln 1-5 zerlegt werden.

Beispiel 1 Als Beispiele diene die bereits weiter oben verwendete Regel:

`search` Partition p `register` p
`where` (p.theme.themeName = 'Hotels' AND p.cardinality > '1000')

Diese Regel wird in drei Schritten in atomare Teilregeln zerlegt. Zuerst löst man die Konjunktion auf (5. Zerlegungsregel):

RegelA: `search` Partition p `register` p
`where` p.theme.themeName = 'Hotels'
RegelB: `search` Partition p `register` p `where` p.cardinality > '1000'
RegelC: `search` RegelA a, RegelB b `register` a `where` a = b

RegelB ist eine auslösende atomare Teilregel, RegelC eine atomare Jointeilregel. Beide können nicht mehr weiter zerlegt werden. RegelA enthält einen Pfadausdruck und muss deshalb entsprechend der 4. Zerlegungsregel aufgeteilt werden (mit Class(p.theme) = Theme):

RegelD: `search` Theme t `register` t `where` t.themeName = 'Hotels'
RegelE: `search` Partition p, RegelD d `register` p `where` p.theme = d

RegelD ist eine auslösende atomare Teilregel, RegelE ist eine Jointeilregel, die in ihrer `search`-Klausel eine Klasse referenziert. Sie muss entsprechend der 6. Zerlegungsregel aufgeteilt werden:

RegelF: `search` Partition p `register` p
RegelG: `search` RegelD d `register` d
RegelH: `search` RegelF f, RegelG g `register` f `where` f.theme = d

Insgesamt ergeben sich also die atomaren Teilregeln:

RegelB: `search` Partition p `register` p `where` p.cardinality > '1000'
RegelC: `search` RegelH h, RegelB b `register` h `where` h = b
RegelD: `search` Theme t `register` t `where` t.themeName = 'Hotels'
RegelF: `search` Partition p `register` p
RegelG: `search` RegelD d `register` d
RegelH: `search` RegelF f, RegelG g `register` f `where` f.theme = g

Erstellung eines Abhängigkeitsgraphen Der Abhängigkeitsgraph repräsentiert *alle* atomaren Teilregeln und die Abhängigkeiten, die bei der Zerlegung der registrierten Regeln entstanden sind. Jede Kante bedeutet, dass die atomare Teilregel des Zielknotens der Kante das Ergebnis der atomaren Teilregel des Quellknotens benötigt.

Bei der Zerlegung wird für jede Regel ein Abhängigkeitsgraph aufgebaut. Dieser wird anschließend in einen globalen Abhängigkeitsgraphen eingefügt, der alle bereits zerlegten Regeln enthält. Gleiche Teilgraphen werden zusammengefasst, dadurch müssen gleiche Teilregeln nur einmal ausgewertet werden.

Jeder Knoten kann noch folgende Annotationen besitzen:

isTrigger: Knoten von auslösenden Teilregeln besitzen diese Annotation. Solche Knoten sind Blattknoten im Abhängigkeitsgraphen. Für eine solche Teilregel muss beim Registrieren überprüft werden, ob sie neu ausgewertet werden muss.
Knoten, die keine Blattknoten sind, müssen nur neu ausgewertet werden, wenn sich das Ergebnis einer Teilregel ändert, von der sie abhängig sind.

isTerminal: Das Ergebnis der Teilregel eines solchen Knotens ist ein Endergebnis, d. h. die Teilregel liefert das Ergebnis einer registrierten Regel. Diese Teilregel (auch Endteilregel genannt) bildet die Wurzel des Graphen, der durch die Zerlegung einer registrierten Regel in atomare Teilregeln entstanden ist.

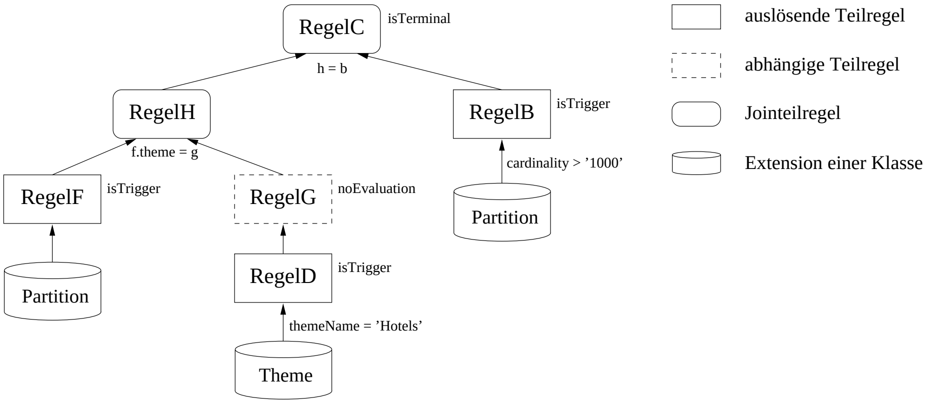

Abbildung 4. Abhängigkeitsgraph für die Regel aus Beispiel 1

isMaterialized: Eine Teilregel mit dieser Annotation materialisiert die Ergebnistupel, die bei der Auswertung dieser Teilregel berechnet werden.
Diese Annotation dient nur der Leistungssteigerung, etwa bei der Auswertung von Jointeilregeln nach einer Änderung. Für die Korrektheit des Algorithmus ist sie nicht notwendig.

noEvaluation: Die Teilregel dieses Knotens darf nur von einer anderen Teilregel verwendet werden und kann keine Endteilregel sein. Wenn sie ausgelöst wird, löst sie sofort die abhängige Teilregel aus, die die Bedingung dieser Teilregel mitprüfen muss. Dadurch entfällt die Erzeugung von unnötigen Zwischenergebnissen.

Beispiel 2 Abbildung 4 zeigt den Abhängigkeitsgraphen, der sich aus den atomaren Teilregeln von Beispiel 1 ergibt. Rechts neben jedem Knoten des Abhängigkeitsgraphen stehen die gesetzten Annotationen. Unterhalb der Jointeilregel-Knoten sind die Joinprädikate ($f.theme = g$ und $h = b$) und unter den auslösenden Teilregel-Knoten sind die Selektionsprädikate ($cardinality > \ '1000'$ und $themeName = \ 'Hotels'$) notiert.

Erzeugung der Vorfiltertabellen Aus den Regeln, den atomaren Teilregeln und dem globalen Abhängigkeitsgraphen werden relationale Tabellen für den Vorfilter erzeugt. Wenn eine L-MDV Metadaten abonniert oder ihre Regelmenge modifiziert, werden diese Tabellen angepasst. Aus Platzgründen werden in dieser Arbeit nur die Tabellen erläutert, die für den Vorfilteralgorithmus benötigt werden.

Für jeden der Vergleichsoperatoren `=`, `>`, `<` und `contains` wird eine eigene Tabelle mit folgendem Schema erzeugt:

$$\begin{aligned}\text{Prefilterrules[EQ|LT|GT|CON]} : \{&[\text{query\_id} : \text{integer}, \text{class} : \text{varchar},\\ &\text{predicate} : \text{varchar}, \text{value} : \text{varchar}]\}\end{aligned}$$

Eine auslösende Teilregel wird abhängig vom Vergleichsoperator, der in ihr verwendet wird, in die entsprechende Tabelle eingefügt. Das Attribut `query_id` enthält die ID der Teilregel, `class` bezeichnet den Namen der Klasse, die in der Teilregel referenziert wird. Die Spalte `predicate` enthält den Namen des Attributs, auf das sich die Teilregel bezieht, und `value` speichert die Stringkonstante, auf die der Operator angewendet wird. Folgende Tabelle zeigt dies für RegelB aus Beispiel 1:

PrefilterrulesGT			
query_id	class	predicate	value
1	Partition	cardinality	1000

Atomare auslösende Teilregeln ohne `where`-Teil werden in einer eigenen Tabelle mit folgendem, vereinfachtem Schema gespeichert:

$$\text{Prefilterrules} : \{[\text{query\_id} : \text{integer}, \text{class} : \text{varchar}\ ]\}$$

Die Prefilterrules-Tabellen dienen als Indexe auf die eingetragenen Regeln, um effizient zu gegebenem Attribut und Wert alle relevanten auslösenden Teilregeln zu finden. Ein ähnlicher Ansatz wird in [PFLS00] verwendet, allerdings im Rahmen eines reinen Hauptspeicheralgorithmus.

Bei der Registrierung von Metadaten trägt der Vorfilteralgorithmus die Objekte des registrierten `RDF`-Dokuments in die Tabelle Prefilterdata ein. Die Tabelle besitzt folgendes Schema:

$$\begin{aligned}\text{Prefilterdata} : \{[&\text{object\_id} : \text{varchar}, \text{class} : \text{varchar},\\ &\text{predicate} : \text{varchar}, \text{value} : \text{varchar}]\}\end{aligned}$$

In diese Tabelle wird für jedes Objekt ein Tupel eingefügt, das die Objekt-ID sowie den Klassennamen enthält. (Die Attribute `predicate` und `value` werden auf `rdf#type` bzw. den Klassennamen gesetzt.) Für jedes Attribut eines Objekts wird ein Tupel mit Objekt-ID, Klassenname des Objekts, Attributname und Attributwert eingetragen.

Beispiel 3 Folgender Auszug aus einem ObjectGlobe-`RDF`-Dokument

```
<Partition rdf:ID="HotelBookPartition">
  <cardinality>30000</cardinality>
</Partition>
```

wird von der MDV in diese Prefilterdata-Tabelle umgewandelt:

object_id	class	predicate	value
...#HotelBookPartition	Partition	rdf#type	Partition
...#HotelBookPartition	Partition	cardinality	30000

Ermittlung relevanter auslösender Teilregeln Mit *einem* Join der Prefilterdata-Tabelle mit allen Prefilterrules-Tabellen bestimmt man alle auslösenden Teilregeln, die von einer Registrierung betroffen sind. Der Join übernimmt dabei nur auslösende Teilregeln, deren Bedingungen erfüllt sind. Aus Gründen der Übersichtlichkeit beschränkt sich die folgende Anfrage auf die PrefilterrulesEQ-Tabelle. Die Tabelle Subclasslist wird verwendet, um Vererbung zu berücksichtigen. Diese ist in `RDF`-Schemata möglich und wird von der MDV unterstützt. Die Tabelle speichert zu jeder Klasse alle im RDF-Schema definierten Unterklassen. Folgende SQL-Anfrage bestimmt alle betroffenen auslösenden Teilregeln und fügt ihre ID und die ID des auslösenden Objekts in die Tabelle Tempviews ein:

```
insert into Tempviews
select r.query_id, d.object_id
from Prefilterdata d, PrefilterrulesEQ r, Subclasslist s
where r.class = d.class and r.predicate = d.predicate
      and r.value = d.value
      and r.type = s.baseclass and d.type = s.subclass
```

Iteratives Auswerten aller relevanten Teilregeln In einer Schleife werden abhängige Teilregeln solange ausgewertet, bis die Tabelle Tempviews leer ist, d. h. bis keine abhängigen Teilregeln mehr ausgelöst werden. Anschließend kann das Endergebnis generiert werden. Jeder Schleifendurchlauf besteht aus folgenden Schritten:

1. **Materialisierung von Auswertungsergebnissen:** Jedes Tupel aus der Tabelle Tempviews, dessen zugehörige Teilregel mit *isMaterialized* annotiert ist, wird materialisiert.
2. **Sichern der Tabelle Tempviews:** Die Tabelle Tempviews wird am Ende des Durchlaufs mit neuen Daten gefüllt, deshalb werden die Tupel von Tempviews in der Tabelle Resultviews gesammelt.
3. **Auswerten abhängiger Teilregeln:** Entsprechend dem Abhängigkeitsgraphen werden nun diejenigen Teilregeln ausgewertet, die durch Teilregeln ausgelöst werden, die in der Tabelle Tempviews enthalten sind. Dazu wird der Regeltext jeder abhängigen Teilregel in SQL umgewandelt und ausgeführt (auf den materialisierten Daten – wenn nötig – und der Tabelle Resultviews). Ergebnistupel (ID des Ergebnisobjekts und Teilregel-ID) werden in die Tabelle Tempviews geschrieben.

Der Abhängigkeitsgraph, aus dem die Prefilterrules-Tabellen erzeugt werden, ist azyklisch und seine Höhe ist endlich. Der Algorithmus terminiert also, da mit jedem Schleifendurchlauf – ausgehend von den Blättern des Graphen – mindestens eine Ebene des Graphen abgearbeitet wird. Zur Bestimmung der Ergebnisse nach der Termination werden aus der Tabelle Resultviews alle Objekt-IDs ausgelesen, die durch eine Endteilregel eingetragen wurden. Die zu den Objekt-IDs gehörenden Objekte werden an alle L-MDVs übertragen, die die entsprechende Regel registriert haben.

4 Der Einsatz der MDV im Sicherheitssystem von ObjectGlobe

4.1 Das Sicherheitssystem von ObjectGlobe

Das ObjectGlobe-System delegiert jede Sicherheitsüberprüfung bezüglich Authentifizierung und Autorisierung an einen speziellen Anbieter, genannt *Security-Provider*. Ein Security-Provider verwendet eine MDV, um Sicherheitsinformationen als *lokale Metadaten* zu speichern[4] – unabhängig von den sonstigen dort gepufferten öffentlichen Metadaten.

ObjectGlobe verwendet als Zugriffskontrollmodell die rollenbasierte Zugriffskontrolle (RBAC) [SCFY96]. Die Administration des Sicherheitssystems übernimmt ein lokaler Sicherheitsadministrator. Dieser ist dafür verantwortlich, Benutzer einzurichten, ihnen Rollen zuzuweisen, usw. Für den Security-Provider wurde das Metadaten-Schema von ObjectGlobe erweitert, um Sicherheitsinformationen speichern zu können [KK00]. Folgender Auszug zeigt die Sicherheitsinformationen eines Benutzers *Otto Normal* im `RDF`-Format[5]:

```
<User rdf:ID="Normal" userName="Otto Normal">
  <authenticationType>password</authenticationType>
  <authenticationData>qZ8uiXFEePpGu</authenticationData>
  <assignedRoles><rdf:Bag>
      <rdf:li rdf:resource="file:/home/og/RBAC.rdf#Manager" />
  </rdf:Bag></assignedRoles>
</User>
```

Jeder Provider des ObjectGlobe-Systems befragt vor der lokalen Instanziierung eines Anfrageplans den ihm zugeordneten Security-Provider, ob der im Plan angegebene Benutzer authentifiziert werden kann und ob er die notwendigen Rechte zur Ausführung des lokalen Teilplans besitzt. Folgende Überprüfungen nimmt ein Security-Provider vor der lokalen Instanziierung eines Anfrageplans im Einzelnen vor:

Authentifizierung des Benutzers: Jeder Anfrageplan ist gegebenenfalls mit Authentifizierungsinformationen des Benutzers annotiert, der die Anfrage gestartet hat. Der Security-Provider entnimmt dem Anfrageplan diese Informationen (Name des Benutzers und Signatur des Plans). Er fordert von der MDV die zum Namen passenden Authentifizierungsdaten an und verifiziert damit die mitgelieferte Signatur. Sind die beiden Signaturen gleich, ist der Benutzer authentifiziert. Ansonsten wird der Plan abgewiesen.

Autorisierung des Plans: Der Security-Provider bestimmt aus dem Teilplan, der lokal ausgeführt werden soll, eine Liste von Rechten, die der Benutzer benötigt. Für den Beispielplan aus Abschnitt 2.2 benötigt der Benutzer *Otto Normal* auf Cycle-Provider *A* folgende Rechte:

[4] Statt der MDV könnten prinzipiell auch andere Systeme verwendet werden, z. B. ein RDBMS oder ein LDAP-Repository [WHK97].

[5] Passwörter (Inhalt der Property `authenticationData`) werden verschlüsselt abgespeichert.

```
<ExecutePermission operatorName="iterators.NestedLoops" />
<ExecutePermission operatorName="iterators.Scan" />
<ReadPermission><partitionName>R</partitionName></ReadPermission>
<ReadPermission wrapperName="wrappers.wrap_S">
  <partitionName>S</partitionName>
</ReadPermission>
```

Eine `ReadPermission` (Recht zum Lesen einer Datenquelle) schließt dabei stets eine `ExecutePermission` (Recht zum Ausführen eines Operators) für die Ausführung des dazugehörigen Wrappers ein.
Der Security-Provider vergleicht die benötigten Rechte mit den Rechten, die für den Benutzer in der MDV gespeichert sind. Folgende Anfrage bestimmt beispielsweise, ob der Benutzer *Otto Normal* das Recht besitzt, den Wrapper *wrappers.wrap_S* auszuführen, um die Datenquelle S zu lesen:

```
search User u, Role r, ReadPermission p select p
where u.userName='Otto Normal' AND
      u.assignedRoles.?=r AND r.assignedPermissions.?=p AND
      p.partitionName='S' AND p.wrapperName='wrappers.wrap_S'
```

Fehlt ein benötigtes Recht, wird die Anfrage abgelehnt.

4.2 Externe Sicherheitssysteme

Wrapper ermöglichen in ObjectGlobe den Zugriff auf externe Datenquellen, die über ein eigenes Sicherheitssystem verfügen können (etwa im Falle eines RDBMS oder eines Web-Servers). ObjectGlobe unterstützt die Übergabe von Authentifizierungsinformationen aus einem Anfrageplan an ein externes Sicherheitssystem, sodass Authentifizierung und Autorisierung bezüglich des Benutzers von diesem externen Sicherheitssystem übernommen werden können.

4.3 Optimierung und Sicherheit

Bei der Anfrageoptimierung benötigt ein ObjectGlobe-Optimierer Sicherheitsanforderungen, d. h. Informationen darüber, auf welche Datenquellen ein Benutzer zugreifen, auf welchen Cycle-Providern er Pläne ausführen und welche Anfrageoperatoren er laden darf. Er benötigt diese Anforderungen von *allen* Providern, die bei der Generierung des Plans berücksichtigt werden sollen.

ObjectGlobe verwendet den Publish/Subscribe-Mechanismus der MDV, um Sicherheitsanforderungen zu verteilen. In diesem Fall bildet die MDV des Security-Providers ein eigenes Backbone. Abonnenten sind die MDVs, die von Optimierern genutzt werden. Die Regeln zum Abonnement von Sicherheitsanforderungen werden nicht von den Abonnenten selbst vorgegeben, wie es beim Abonnement von öffentlichen Metadaten der Fall ist, sondern vom Administrator des Security-Providers, dessen Sicherheitsanforderungen abonniert werden können. Die Entscheidung darüber, welche Sicherheitsanforderungen abonniert werden dürfen, müssen die Anbieter von Security-Provider und Optimierer aushandeln. Sie wird hauptsächlich von der unternehmensinternen Sicherheitspolitik des Security-Provider-Anbieters bestimmt.

Beispiel 4 Folgende beiden Regeln abonnieren sämtliche Daten des Benutzers *Otto Normal* (inklusive seiner Rechte):

- `search` User u `register` u `where` u.userName = 'Otto Normal'
- `search` Permission p, Role r, User u `register` p
  `where` u.userName='Otto Normal' `AND`
  u.assignedRoles.?=r `AND` r.assignedPermissions.?=p

Änderungen, die diesen Benutzer betreffen, etwa Änderungen an den ihm zugewiesenen Rechten, werden damit automatisch an den Abonnenten propagiert.

4.4 Sicherheitsanforderungen externer Sicherheitssysteme

Bei externen Sicherheitssystemen ist zu beachten, dass sich deren Sicherheitsanforderungen initial nicht in der MDV befinden. Änderungen an den Daten finden *nicht* in der MDV, sondern im externen Sicherheitssystem statt. ObjectGlobe unterstützt die Integration derartiger Sicherheitsanforderungen durch das Java-Interface `Extractor`, das die zur Extraktion von Sicherheitsanforderungen notwendige Funktionalität definiert. Jeder Provider, der ObjectGlobe die Extraktion seiner Sicherheitsanforderungen ermöglichen will, muss eine Klasse zur Verfügung stellen, die dieses Interface geeignet implementiert (analog der Implementierung des Wrapper-Interfaces zur Integration einer Datenquelle).

Beim erstmaligen Zugriff auf einen Provider wird der passende Extractor (falls vorhanden) benutzt, um sämtliche verfügbaren Sicherheitsanforderungen aus dem externen Sicherheitssystem zu extrahieren und in der MDV des Security-Providers zu speichern. Durch den Publish/Subscribe-Mechanismus gelangen die Daten zu den MDVs, die von den Optimierern benutzt werden. Änderungen an den externen Sicherheitsanforderungen werden auf folgende Arten erkannt:

1. Die Anfrage eines Wrappers wird vom externen Sicherheitssystem der Wrapper-Datenquelle abgelehnt. Der Optimierer hat also offensichtlich die Anfrage mit veralteten Sicherheitsanforderungen generiert. Der Wrapper meldet diese Tatsache an den ihn ausführenden Cycle-Provider, der zum einen den Abbruch der Anfrage veranlasst, zum anderen den passenden Extractor benutzt, um die geänderten Sicherheitsanforderungen aus dem externen Sicherheitssystem zu extrahieren und die lokalen Metadaten der MDV anzupassen.
2. Der Administrator des externen Sicherheitssystems benachrichtigt die Cycle-Provider, die per Wrapper auf den Provider zugreifen dürfen, von der Änderung. Per Extractor extrahieren diese die geänderten Daten und integrieren sie in die MDV ihres Security-Providers.
3. Ein Cycle-Provider überprüft in periodischen Abständen mit dem passenden Extractor, ob sich im externen Sicherheitssystem Daten geändert haben.

5 Zusammenfassung

In dieser Arbeit haben wir die verteilte Metadatenverwaltung MDV präsentiert, die im Rahmen von ObjectGlobe, einem offenen und verteilten Anfragebearbeitungssystem für Internet-Datenquellen, entwickelt wurde. Wir haben zuerst ihre verteilte Architektur erläutert, die aus wenigen öffentlichen MDVs und vielen lokalen MDVs besteht und eine effiziente Auswertung von lokalen Anfragen erlaubt. Die Registrierung von Metadaten im `RDF`-Format erfolgt an den öffentlichen MDVs. Durch eine geeignete Regelmenge abonnieren lokale MDVs relevante Metadaten basierend auf dem Inhalt der Daten und puffern diese lokal. Die Auswertung von Anfragen an die Metadaten erfolgt lokal, um unnötige Verzögerungen durch Kommunikation über das Internet zu vermeiden. Die Aktualität und Konsistenz der gepufferten Metadaten gewährleistet ein Publish/Subscribe-Mechanismus, der bei der Modifikation von Metadaten automatisch die entsprechenden Abonnenten benachrichtigt. Detailliert haben wir den mit bestehender relationaler DB-Technologie implementierten Vorfilter der MDV beschrieben, der aus den registrierten Regeln bestimmt, welche lokalen MDVs von modifizierten Metadaten benachrichtigt werden müssen.

Die MDV wird in ObjectGlobe nicht nur zur Registrierung von Ressourcen eingesetzt, sondern auch zur Verwaltung von Sicherheitsanforderungen, insbesondere haben wir ihre Verwendung bei der Verteilung von lokalen Sicherheitsinformationen beschrieben. Dabei werden die lokalen Daten aus externen Sicherheitssystemen extrahiert und in lokale MDVs integriert. Der Publish/Subscribe-Mechanismus überträgt sie an andere MDVs, sodass sie bei der Anfragebearbeitung auf Internet-Datenquellen verwendet werden können. Anschließend haben wir verschiedene Möglichkeiten aufgezeigt, um Änderungen an externen Sicherheitsinformationen zu erkennen und die Daten der lokalen MDVs automatisch anzupassen.

Das ObjectGlobe-Projekt, in dessen Rahmen die hier beschriebene MDV entstand, startete vor etwa drei Jahren. Eine erste Demonstration (über unsere WWW-Seiten erreichbar) wurde auf der SIGMOD 99 [BKK99b] vorgeführt. Diese stellt eine vereinfachte E-Commerce-Plattform für Reiseagenturen dar. Die MDV wurde im Rahmen einer Diplomarbeit implementiert, die Integration in das ObjectGlobe-System wurde vor kurzem abgeschlossen. Für die Zukunft sind ausführliche Benchmarkmessungen geplant, um die Skalierbarkeit der MDV und die Effizienz des Vorfilteralgorithmus zu untersuchen.

Literatur

[AF00] M. Altinel und M. J. Franklin: *Efficient Filtering of XML Documents for Selective Dissemination of Information.* In: *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Seiten 53–64, Cairo, Egypt, September 2000.

[Arn99] K. Arnold: *The Jini(TM) Specification (The Jini(TM) Technology Series).* Addison-Wesley, Reading, MA, USA, 1999.

[ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley und T. D. Chandra: *Matching Events in a Content-Based Subscription System.* In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, Seiten 53–61. ACM, 1999.

[BG99] D. Brickley und R. V. Guha: *Resource Description Framework (RDF) Schema Specification.* `http://www.w3.org`, March 1999.

[BKK+99a] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam und K. Stocker: *ObjectGlobe: Ubiquitous Query Processing on* *the Internet.* Technischer Bericht MIP-9909, Universität Passau, 94030 Passau, Germany, 1999.

[BKK99b] R. Braumandl, A. Kemper und D. Kossmann: *Database Patchwork on the Internet (Project Demo Description).* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 550–552, Philadelphia, PA, USA, Juni 1999.

[BKK+00] R. Braumandl, M. Keidl, A. Kreutz, S. Seltzsam und K. Stocker: *Das Metadaten-Schema von ObjectGlobe.* `http://www.db.fmi.uni-passau.de/~objglobe/ObjectGlobe-Metaschema.rdf`, 2000.

[BLT86] J. A. Blakeley, P.-Å. Larson und F. W. Tompa: *Efficiently Updating Materialized Views.* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 61–71, Washington, D.C., 1986.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian und Y. Wang: *NiagaraCQ: A Scalable Continuous Query System for Internet Databases.* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 379–390, Dallas, Texas, USA, Juni 2000.

[CSS99] S. Conrad, G. Saake und K.-U. Sattler: *Informationsfusion - Herausforderung an die Datenbanktechnologie.* In: *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications (BTW)*, Informatik aktuell, New York, Berlin, etc., 1999. Springer-Verlag.

[CZH+99] S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph und R. H. Katz: *An Architecture for a Secure Service Discovery Service.* In: *Proc. of ACM MOBICOM Conference*, Seiten 24–35, Seattle, WA, August 1999.

[GMS93] A. Gupta, I. S. Mumick und V. S. Subrahmanian: *Maintaining Views Incrementally.* In: P. Buneman und S. Jajodia (Herausgeber): *Proc. of the ACM SIGMOD Conf. on Management of Data*, Band 22, Seiten 157–166, Mai 1993.

[GPVD99] E. Guttman, C. Perkins, J. Veizades und M. Day: *Service Location Protocol, V. 2.* `http://www.rfc-editor.org/rfc/rfc2608.txt`, Juni 1999.

[Gra93] G. Graefe: *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, 25(2):73–170, Juni 1993.

[Han87] E. Hanson: *A Performance Analysis of View Materialization Strategies.* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 440–453, San Francisco, CA, Mai 1987.

[JD93] D. Jonscher und K.R. Dittrich: *Access Control for Database Federations: a discussion of the state-of-the-art.* In: *Proc. DBTA Workshop on Interoperability of Database Systems and Database Applications*, 1993.

[JD95] D. Jonscher und K.R. Dittrich: *Argos - A Configurable Access Control System for Interoperable Environments.* In: D. L. Spooner, S. A. Demurjian und J. E. Dobson (Herausgeber): *Database Security IX: Status and Prospectus*, Amsterdam, 1995. Elsevier Science Publishers B.V.

[KK00] M. Keidl und A. Kreutz: *Das Metadaten-Schema des Security-Providers von ObjectGlobe.* `http://www.db.fmi.uni-passau.de/~objglobe/ObjectGlobe-Metaschema-SecurityProvider.rdf`, 2000.

[LHM+86] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh und P. F. Wilms: *A Snapshot Differential Refresh Algorithm.* Seiten 53–60, 1986.

[LPT99] L. Liu, C. Pu und W. Tang: *Continual Queries for Internet Scale Event-Driven Information Delivery.* IEEE Transactions on Knowledge and Data Engineering, 11(4):610–628, 1999.

[LS99] O. Lassila und R. R. Swick: *Resource Description Framework (RDF) Model and Syntax Specification.* `http://www.w3.org`, Feb 1999.

[MRT98] G. A. Mihaila, L. Raschid und A. Tomasic: *Equal Time for Data on the Internet with WebSemantics.* In: *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, Band 1377 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 87–101, Valencia, Spain, März 1998. Springer-Verlag.

[Per92] G. Pernul: *Canonical Security Modeling for Federated Databases.* In: *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems*, Seiten 207–222, 1992.

[PFLS00] J. Pereira, F. Fabret, F. Llirbat und D. Shasha: *Efficient matching for web-based publish/subscribe systems.* In: *Proc. of the IECIS International Conference on Cooperative Information Systems*, Israel, September 2000.

[RMR00] M. Rodriguez-Martinez und N. Roussopoulos: *Automatic Deployment of Application-Specific Metadata and Code in MOCHA.* In: *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, Band 1777 der Reihe *Lecture Notes in Computer Science (LNCS)*, Konstanz, Germany, März 2000. Springer-Verlag.

[SAP99] SAP: *Business Networking in the Internet Age.* Technischer Bericht SAP White Paper, Sep 1999. `http://www.sap-ag.de/germany/products/my-sap/pdf/bus_networking.pdf`.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein und C. E. Youman: *Role-Based Access Control Models.* IEEE Computer, 29(2):38–47, 1996.

[SL90] A. Sheth und J. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonmous Databases.* ACM Computing Surveys, 22(3):183–236, September 1990.

[SST+99] K. Schwarz, I. Schmitt, C. Türker, M. Höding, E. Hildebrandt, S. Balko, S. Conrad und G. Saake: *Design Support for Database Federations.* In: *Conceptual Modeling - ER '99, 18th Int. Conf. on Conceptual Modeling*, Band 1728 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 445–459. Springer-Verlag, November 1999.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols und B. M. Oki: *Continuous Queries over Append-Only Databases.* In: *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seiten 321–330, San Diego, CA, USA, Juni 1992.

[TLW87] M. Templeton, E. Lund und P. Ward: *Pragmatics of Access Control in Mermaid.* IEEE Data Engineering Bulletin, 10(3):33–38, 1987.

[UDD00] *UDDI Technical White Paper.* White Paper, Ariba, Inc., IBM Corp., and Microsoft Corp., September 2000. `http://www.uddi.org/`.

[UPN00] *Universal Plug and Play Device Architecture.* Document, Microsoft Corporation, Juni 2000. `http://www.upnp.org`.

[WHK97] M. Wahl, T. Howes und S. Kille: *Lightweight Directory Access Protocol (v3).* `ftp://ftp.isi.edu/in-notes/rfc2251.txt`, Dezember 1997.

[WL93] T. Y. C. Woo und S. S. Lam: *Authorizations in Distributed Systems: A New Approach.* Journal of Computer Security, 2(2-3):107–136, 1993.

[WS87] C.-Y. Wang und D. L. Spooner: *Access Control in a Heterogeneous Distributed Database Management.* In: *Proc. of the Symposium on Reliability in Distributed Software and Database Systems*, Seiten 84–92, März 1987.

[YGM99] T. W. Yan und H. Garcia-Molina: *The SIFT Information Dissemination System.* ACM Trans. on Database Systems, 24(4):529–565, 1999.