Prüfer:	Prof. Mitschang
Betreuer:	Thomas Schwarz
begonnen am	1.11.2003
beendet am	16.5.2004
unterbrochen um	2 Wochen
CR-Klassifikation	C.2.4,H.3.3,H.3.5

Diplomarbeit Nr. 2164

Operator-basierte Anfrageverarbeitung in der Nexus Föderation

Steffen Motzer

Kurzfassung

Die Forschergruppe Nexus entwickelt eine offene Plattform für mobile, ortsbezogene Anwendungen. Durch die Offenheit können beliebige Datenanbieter ihre Daten durch die Plattform bereitstellen. Eine Föderationsmiddleware soll die Verteilung der Daten vor einer Anwendung verbergen und die Daten verschiedener Anbieter in geeigneter Weise kombinieren.

Anfragen werden bisher in der Föderation nach einem hart-codierten Plan verarbeitet. Dies soll nun flexibilisiert werden, indem die Schritte des bisherigen Plans in einzelne Operatoren verpackt werden, welche für jede Anfrage dynamisch ausgewählt, konfiguriert und kombiniert werden können.

Im Rahmen dieser Diplomarbeit wurde die Föderation neu implementiert, wobei die einzelnen Schritte des bisherigen Plans in Operatoren ausgeführt werden. Die im Rahmen dieser Diplomarbeit implementierte Föderation verarbeitet den restriction, aber nicht den closest Teil einer Anfrage in AWQL.

1 Einleitung

1.1 Hintergrund

In den letzten Jahren besitzen immer mehr Menschen mobile Geräte wie Handys oder PDAs. Auch drahtlose LANs, die teilweise einen Zugang zum Internet ermöglichen, sind an immer mehr Orten verfügbar. Mobile Geräte, wie PDAs oder Handys sind oft dazu in der Lage, ihre Position über GPS oder über die Basisstationen (im GSM Netz) zu bestimmen.

Diese verfügbaren Informationen können zum Beispiel wie im folgenden Szenario praktisch verwendet werden:

Ein Tourist kommt am Stuttgarter Hauptbahnhof mit dem Taxi an. Sein PDA, der in einem verteilten System registriert ist, meldet seine aktuelle Position an dieses System. Sobald sich der PDA im Bahnhof befindet und von einem Server des Bahnhofs dort gefunden wurde, fragt dieser Server nach den Attributen des PDAs wie die Sprache, den Namen des Besitzers und eine Adresse an die Nachrichten gesendet werden könnten. Der Server erkennt den PDA des Touristen und findet das Ticket und die Platzreservierung des Kunden im Buchungssystem. Daraufhin sendet der Server eine Nachricht an den PDA und begrüßt den Kunden und informiert ihn über eine mögliche Verspätung des Zugs.

Am Zielort angekommen, möchte der Tourist nachdem er ein Hotel gefunden hat, Essen gehen. Da er sich nicht in der Stadt auskennt, sucht er mit seinem PDA nach einem Stadtplan, welcher von der Routenplaneranwendung seines PDA verwendet werden kann. Der PDA startet eine Anfrage und erhält mehrere Angebote. Der Tourist wählt den kostenlosen Stadtplan des Öffentlichen Nahverkehrsunternehmens aus, welcher auch Informationen über Bus und S-Bahnverbindungen enthält. Er startet eine Anfrage für das Objekt und erhält es. Da er auch keine Restaurants in dieser Stadt kennt, startet er jetzt eine Anfrage nach einem China Restaurant. Die gefundenen Restaurants sind allerdings entweder zu weit entfernt oder nicht günstig mit öffentlichen Verkehrsmitteln zu erreichen. Also startet er eine weitere Anfrage nach allen Restaurants. Er erhält neben den bisher gefundenen Restaurants auch ein Objekt eines gut mit dem Bus erreichbaren Bistros. Der Routenplaner des PDA verwendet den zuvor gefundenen Stadtplan, um dem Touristen den Weg zu erklären und bezahlt bei Benutzung des Busses automatisch das Ticket.

1.2 Motivation

In dem Beispiel wurde nach Informationen gesucht, welche von einer Vielzahl von Servern bereitgestellt wird. Im Nexus System können Server von Anderen eingefügt werden. Die einzelnen Anbieter von Daten wollen die Kontrolle über die eigenen Objekte behalten und diese auf den eigenen Servern verwalten. Ein Restaurant zum Beispiel will eine Speisekarte zur Werbung im Nexus System auf einem eigenen Server verfügbar machen und ein Abieter von Stadtplänen will diese aus Urheberrechtsgründen nur auf seinem eigenen Server verbreiten. Die von den Servern verwalteten Objekte sind dabei in der Regel auf bestimmte Objekt-Typen und ein bestimmtes Gebiet beschränkt.

Auch andere Datenquellen, wie zum Beispiel WWW-Server oder Sensoren (Wasserstandsmelder an einem Fluss) könnten in das Nexus System eingefügt werden. Die Menge aller Daten ist viel zu umfangreich, um sie auf einem zentralen Server zu verwalten, zumal dieser dann (ohne Replikation und voneinander unabhängiger Netzwerkanbindung) ein single point of failure des

verteilten Systems währe.

Damit die Anwendungen integrierte Zugriffe auf diese Daten vornehmen können, wird eine Föderation entworfen, welche eine integrierte und einheitliche Sicht auf alle Informationen ermöglicht. Die Aufgabe der Föderation ist es die auf den Servern verteilten Informationen den Anwendungen, welche Anfragen stellen, verfügbar zu machen, ohne daß diese Kenntnis von diesen Servern besitzen müssen. Dabei ermittelt die Föderation die Server, welche eine Teil-Antwort liefern können, stellt für die Anwendung eine Teil-Anfrage, fasst die Antworten der Server zusammen und gibt das Ergebnis der Anwendung zurück.

1.3 Aufgabenstellung

Eine solche Föderation existiert bereits. Um einzelne Komponenten leicht erweitern oder austauschen zu können, soll die Föderation neu implementiert werden. Die einzelnen Schritte der Abarbeitung einer Anfrage sollen in einzelnen Operatoren implementiert werden. Auch weitere noch nicht vorhergesehenen Bearbeitungsschritte können in Operatoren implementiert werden und in die Föderation integriert werden. Operatoren können auch jederzeit einfach neu oder anders implementiert und ausgetauscht werden. Gibt es für einen Operator mehr als eine mögliche Implementierung, dann könnte zur Laufzeit ausgewählt werden, welcher Operator verwendet werden soll.

1.4 Aufbau des Berichts

In Kapitel 2 dieser Diplomarbeit werden zuerst die vorhandenen Komponenten der Nexus Plattform kurz vorgestellt. In Kapitel 4 werden die zu entwerfenden Komponenten der Föderation, die Anforderungen an diese Komponenten, deren notwendigen Eigenschaften und wie die Komponenten miteinander interagieren diskutiert. In Kapitel 5 wird der Grobentwurf der Komponenten vorgestellt. In Kapitel 6 wird der Feinentwurf der Klassen der Komponenten vorgestellt. Danach wird ein Test von einzelnen Komponenten durchgeführt und darauf basierend im Rückblick auf die Diplomarbeit ein Ausblick auf weitere Erweiterungen der Föderation gegeben.

2 Die Nexus Plattform

Um die Föderation neu implementieren zu können, ist es notwendig zuerst die Architektur und die Komponenten der Nexus Plattform vorzustellen. Dazu wird in diesem Kapitel die Vorstellung der Nexus Plattform aus [Nicklas et al 2001] zusammengefaßt wiedergegeben. Die Föderation wird in diesem Kapitel sehr viel ausführlicher vorgestellt, da sie für den Entwurf und die Implementierung sehr viel wichtiger ist als die anderen Komponenten der Nexus Plattform.

2.1 Die Architektur

Die Nexus Plattform bildet ein verteiltes System. Die Komponenten der Plattform können dabei in drei Schichten angeordnet werden:

- Die Nexus Anwendungs-Schicht
- Die Föderationsschicht mit den Nexus Neides und dem *Area Service Register*
- Die Dienste-Schicht.

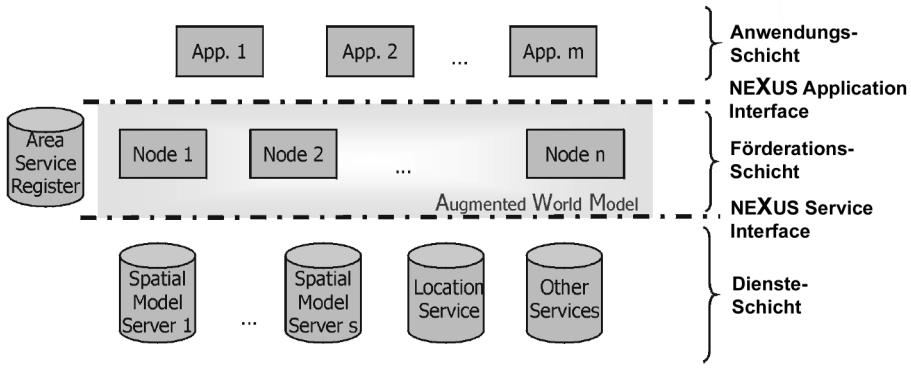


Abb. 1: Die Architektur der Nexus Plattform (nach [Nicklas et al 2001])

Verglichen mit dem World Wide Web entsprechen die Dienste den Webservern. Jede Datenquelle, die der *Augmented World* Objekte liefern will, muß das Nexus Service Interface implementieren und sich bei der Föderation, also dem *Area Service Register* registrieren.

Die Nexus Nodes vermitteln zwischen den Nexus Anwendungen und den Diensten. Die Föderation gibt den Anwendungen eine integrierte Sicht auf das *Augmented World Model* also auf die Abbildung der realen (und der virtuellen) Welt.

2.2 Das Augmented World Model

Das *Augmented World Model* ist ein objektorientiertes Informationsmodell für mobile umgebungsbezogene Anwendungen. Der Sinn dahinter ist, daß die Benutzer solcher Anwendungen in einer Welt mit realen Objekten leben, welche durch virtuelle Objekte ergänzt werden. Diese virtuellen Objekte repräsentieren z.B. externe Informationsräume wie das WWW.

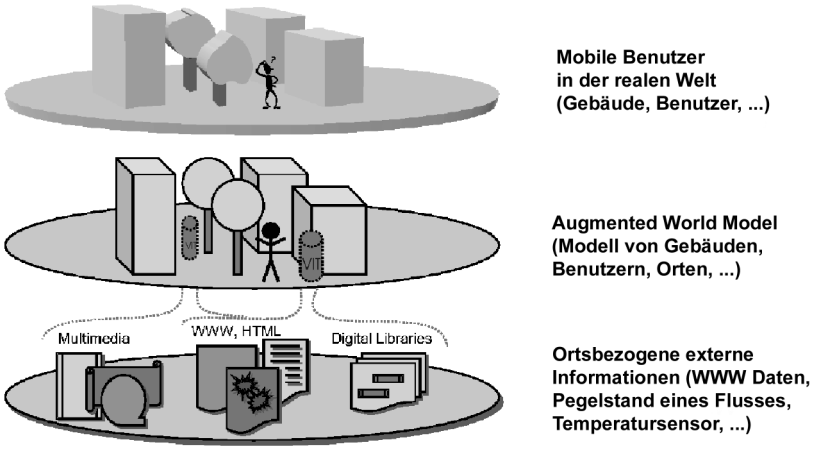


Abb. 2: Die Augmented World der Nexus Plattform (nach [Nicklas et al 2001])

2.2.1 Reale Objekte

Es ist möglich ortsbezogene Anwendungen zu implementieren, ohne Objekte der realen Welt in dem Modell abzubilden. In Nexus allerdings sollen umgebungsbezogene Anwendungen nicht nur ihre eigene Position kennen. Sie sollen auch ihre Umgebung kennen. Zum Beispiel soll ein Anwender auf ein reales Objekt wie eine Bücherei zeigen können, um Informationen wie die Homepage dieser Bücherei zu erhalten. In anderen Fällen besteht eine Beziehung zwischen virtuellen Objekten und realen Objekten. Ein Beispiel dafür ist im Szenario des vorherigen Kapitels das abgerufene Video das sich z.B. auf das reale Objekt, die Sehenswürdigkeit, bezieht.

2.2.2 Virtuelle Objekte

Virtuelle Objekte sind Objekte, die in der realen Welt so nicht vorkommen müssen und repräsentieren externe Informationen die in dem *Augmented World Model* verfügbar gemacht werden wie zum Beispiel der Stadtplan im vorherigen Beispiel in Kapitel 1.

Um die Navigation zu ermöglichen existieren im *Augmented World Model (AWM)* auch virtuelle Navigations-Objekte (Ecken und Nodes), die sich auf reale Objekte wie Kreuzungen und Straßen beziehen.

2.2.3 Das Standard Class Schema

Um zu definieren, welche Objekte es gibt und um eine gemeinsame Sicht bezüglich der Semantik dieser Objekte zu ermöglichen, werden die Klassen der Objekte in Schemata definiert. In einem *Standard Class Schema (SCS)* sind die Attribute definiert, die alle Objekte der Klassen, die von dem SCS erben, enthalten können.

2.2.4 Extended Class Schemata

Die durch das SCS definierten Klassen unterstützen nicht alle möglichen Informationen. Damit nicht alle theoretisch denkbaren Attribute in das SCS aufgenommen werden müssen

(was auch nicht möglich ist, da das SCS dann nie wirklich vollständig wäre), werden *Extended Class Schemata (ECS)* benutzt. Die Klassen dieser Schemata erben von anderen Klassen des ECS und vom SCS. Dabei werden die Attribute vererbt. So könnte zum Beispiel die Klasse *MobilePhone* von der Klasse *mobileobject* erben. Die Klasse *MobilePhone* könnte weitere Attribute hinzufügen, wie zum Beispiel eine Telefonnummer an die SMS-Nachrichten wie eine Gewitterwarnung gesendet werden dürfen.

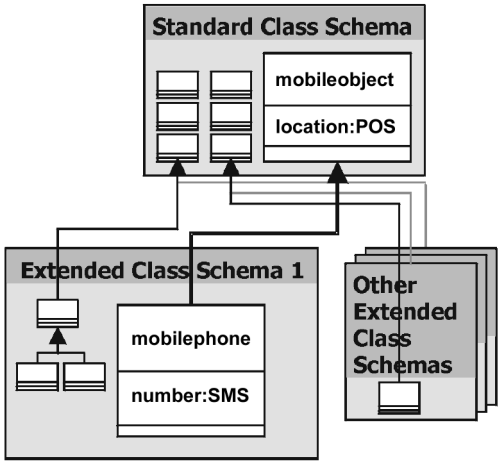


Abb. 3: Standard und Extended Class Schemata (nach [Nicklas et al 2001])

2.2.5 Optionale Attribute

In einem Schema können viele Attribute definiert werden. Oft sind allerdings nicht alle Attribute interessant oder relevant für ein bestimmtes Objekt. Es soll auch möglich sein, daß bestimmte Attribute bei der Registrierung nicht angegeben werden. Deshalb sind die meisten Attribute optional. Sollen sie angegeben werden, dann ist die darunterliegende Semantik bekannt.

Die Adresse eines PDA, an die Nachrichten gesendet werden können, sollte zum Beispiel optional sein um zu verhindern, daß unerwünschte Werbung an den PDA gesendet wird.

2.2.6 Datenaustausch in der Nexus Plattform

Die Komponenten der Nexus Plattform müssen Objekte der AWM untereinander austauschen. Damit diese von einer Komponente an eine andere Komponente übergeben werden können, müssen die Objekte serialisiert werden. Unabhängig von dem auf der jeweiligen Komponente benutzten Betriebssystem, Hardware und Programmiersprache in der die Komponente programmiert wurde, müssen die Objekte ausgetauscht werden können. Deshalb bietet es sicht an, die Objekte in XML auszutauschen. Dafür wurde eine XML Sprache definiert; die *Augmented World Modeling Language (AWML)*.

Auch die Anfragen an Dienste und die Föderation werden in einer XML Sprache übergeben. Diese Sprache ist die *Augmented World Query Language (AWQL)*.

2.3 Die Komponenten der Nexus Plattform

Hier werden die Funktionen der einzelnen Komponenten kurz erläutert.

2.3.1 Die Föderation

Die Föderations-Ebene bietet eine einheitliche Sicht auf die von den Nexus Diensten zur Verfügung gestellten Daten an. Die Nexus Nodes implementieren das *Nexus Application Interface* über das die Anwendungen die Dienste der Föderations-Schicht nutzen können. Die Dienste der Nexus Plattform wie die *Spatial Model Server* müssen das *Nexus Service Interface* implementieren und sich bei dem *Area Service Register* registrieren, damit die Föderation diese Dienste kennt und Anfragen an sie senden kann.

Das *Area Service Register (ASR)* ist ein räumlicher Verzeichnisdienst, den die Föderation zum Auffinden von Diensten verwendet.

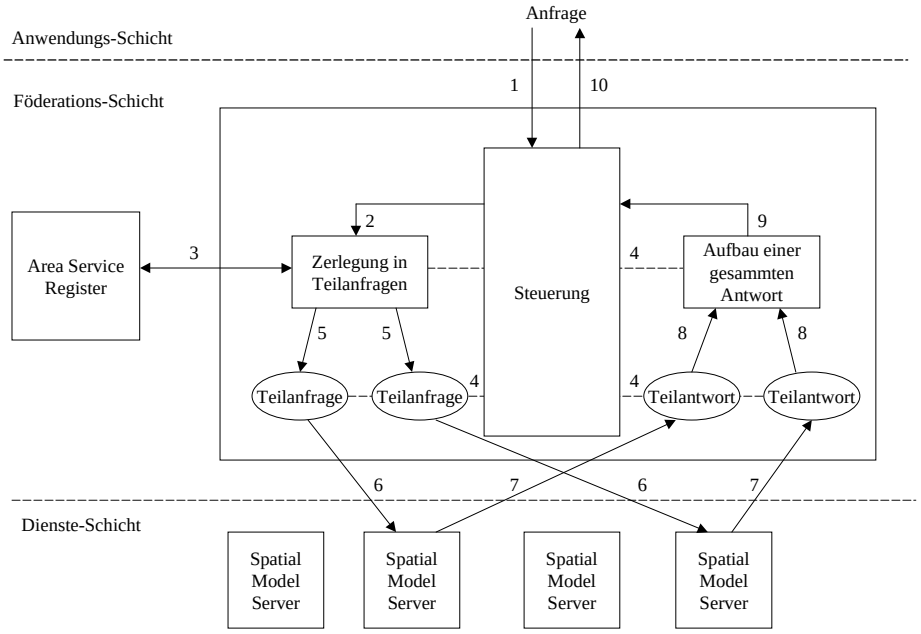


Abb. 4: Beispiel einer Anfrage an die Föderation

In dem obigen Beispiel wird dargestellt, wie die zu entwerfende Föderation eine Anfrage bearbeiten soll. Der zu integrierende Cache wird in diesem Beispiel der Übersichtlichkeit wegen nicht berücksichtigt. Zur Verarbeitung werden folgende Schritte ausgeführt:

1: Die Anfrage wird an die Föderation übergeben. Die Föderation formt die Anfrage intern um (ein *Restriction* Ausdruck wird in eine Distributive Normalform konvertiert).

2: Die Anfrage soll jetzt in Teilanfragen zerlegt werden, wobei jede Teilanfrage an einen anderen Server (oder den Cache) gestellt werden soll.

3: Um die Nexus Server der Diensteschicht zu bestimmen, kann es notwendig sein eine Anfrage an das Area Service Register zu stellen.

4: Ist die Anfrage in Teilanfragen zerlegt, dann erstellt die Steuerung Operatoren, welche die Teilanfragen an die Nexus Server der Dienste-Schicht übergeben, das Ergebnis entgegennehmen, es verarbeiten und zu einer kompletten Ergebnis zusammenfassen.

5: Die Teilanfragen werden den Operatoren übergeben.

6: Die Operatoren übergeben die Anfragen an die Server.

7: Die Server geben die Antwort zurück. Dies ist eine Menge von Nexus Objekten oder ein *Change Response Language* (CRL) Dokument welches angibt, ob die Operation erfolgreich ausgeführt wurde.

8: Die Antworten der Server werden bearbeitet. Konnte ein Server einen Restriction Ausdruck nicht verarbeiten, dann wendet die Föderation diesen *Restriction* Ausdruck jetzt auf die Antwort an. Zum Zeitpunkt dieser Diplomarbeit konnten alle Server einen *Restriction* Ausdruck verarbeiten. Die Antworten werden zu einer konsistenten Antwort kombiniert. Werden Mengen von Nexus Objekten vereinigt und kommt ein Nexus Objekt in mehr als einer Menge vor, dann werden die Objekte miteinander vereinigt. Handelt es sich um mobile Objekte, dann sollten die in beiden Objekten vorhandene Attribute von den Attributen des aktuelleren Objekts überschrieben werden.

9 und 10: Die Antwort wird an die Anwendung zurückgegeben.

2.3.2 Die Area Service Register

Das *Area Service Register* ist ein Register, das für jede *Augmented Area* Informationen speichert. Für jede *Augmented Area* wird das geographische Gebiet, der dafür zuständige Server, die Klassen der von diesem Server verwalteten Objekte und der Name des Servers gespeichert.

Diese Einträge werden von der Föderation dazu benutzt, um die Server (eines Dienstes) zu finden an den eine Anfrage gestellt werden soll.

2.3.3 Die Spatial Model Server

Ein *Spatial Model Server* verwaltet statische Objekte. Ein solcher Dienst muß das *Nexus Service Interface* implementieren und Objekte in der Sprache AWML zurückgeben.

Er muß nicht unbedingt das Ändern von Attributen, das Registrieren oder das Entfernen von Objekten unterstützen, sofern er auf die entsprechenden Anfragen korrekt reagiert. Ein solcher Dienst könnte zum Beispiel an einer bestimmten Position ein Objekt anbieten, das den aktuellen gemessenen Pegelstand eines Flusses angibt.

Solche Objekte müssen lediglich gültige Objekte des AWM sein.

2.3.4 Der Lokationsdienst

Spatial Model Server können keine mobilen Objekte verwalten. Bewegt sich ein mobiles Objekt von einem Dienstgebiet in das eines anderen Servers so muß das Objekt an diesen übergeben werden, was von *Spatial Model Servern* nicht unterstützt wird. Auch wenn sich ein

mobiles Objekt in ein Gebiet bewegt, das von keiner *Augmented Area* überdeckt ist könnte es verloren gehen. Es ist ebenfalls notwendig, daß der Lokationsdienst Lokations-Updates effizient durchführt.

Deshalb wurde ein verteilter Lokationsdienst konzipiert. Die Server des Lokationsdienstes kommunizieren untereinander per UDP. An den Lokationsdienst kann über die *AWQLWrapper* Schnittstelle Anfragen in AWQL gestellt und Objekte per AWML registriert werden.

3 Vorarbeiten

In diesem Kapitel werden zuerst die zu verarbeitenden Datenstrukturen vorgestellt.

Danach werden die von der Föderation benötigten und verwendeten Java-Klassen vorgestellt, welche bereits vorhanden sind und nicht mehr implementiert werden müssen.

Anschließend wird noch Apache Tomcat und Apache SOAP vorgestellt. Apache Tomcat und Apache SOAP werden dazu verwendet, um auf die zu entwerfende Schnittstelle zur Föderation zugreifen zu können.

Zuletzt wird die Anbindung des Cache skizziert und die notwendigen Klassen, welche übergeben werden müssen erläutert.

3.1 Die Datenstrukturen

Hier werden die von der Föderation verarbeiteten Datenstrukturen und Dokumente in XML vorgestellt. Dabei wird auch darauf eingegangen, wie die zu entwerfende Föderation die Datenstrukturen verarbeitet.

3.1.1 AWQL

Suchanfragen an die Föderation werden in der Abfrage-Sprache AWQL 1.1 (*Augmented World Query Language*) gestellt. In der Anfrage wird angegeben, welche Objekte zurückgegeben werden sollen, welche Eigenschaften zurückgegebene Objekte erfüllen müssen und welche Attribute von Objekten zurückgegeben werden sollen. *Query*-Anfragen an *SpatialModelServer* und *AWQLWrapper* werden in AWQL gestellt

Um Attribute oder die Position von registrierten Objekten neu zu setzen oder die Objekte zu entfernen wird ebenfalls eine in AWQL 1.1 kodierte Anfrage an die Föderation übergeben.

Die DTD von AWQL ist nicht mehr gültig. Sie wurde durch ein XML Schema ersetzt und ist hier jedoch angegeben, um einen Überblick über die Eigenschaften von AWQL zu geben:

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<!ELEMENT awql (aas?, scope?, restriction?, closest?,
   (filter?, generalization?, aggregation?) | update)>
<!ENTITY %boolexpr “(and | or | not | equal | in | inside | overlaps)”>
<!ELEMENT nexusdata (#PCDATA)>
<!ELEMENT aas (naal)+>
<!ELEMENT naal (#PCDATA)>
<!ELEMENT scope (ecs)+>
<!ELEMENT ecs EMPTY>
<!ATTLIST ecs name CDATA #REQUIRED is NMTOKEN #IMPLIED>
<!ELEMENT restriction (%boolexpr;)>
<!ELEMENT and (%boolexpr;)+>
<!ELEMENT or (%boolexpr;)+>
<!ELEMENT not (%boolexpr;)>
<!ELEMENT equal (attr nexusdata)>
<!ELEMENT in (attr nexusdata*)>
<!ELEMENT inside (attr nexusdata)>
<!ATTLIST inside acc NMTOKEN #IMPLIED>
<!ELEMENT overlaps (attr nexusdata)>
<!ATTLIST overlaps acc NMTOKEN #IMPLIED>
<!ELEMENT filter (((includes | includeallother)?, excludes?) |
```

```
    (includes?, (excludes | exculdeallother)?))>
<!ELEMENT includes (attr)+>
<!ELEMENT excludes (attr)+>
<!ELEMENT includeallother EMPTY>
<!ELEMENT excludeallother EMPTY>
<!ELEMENT closest (nexusdata)>
<!ATTLIST closest num NMTOKEN “!” acc NMTOKEN #IMPLIED>
<!ELEMENT generalization (#PCDATA)>
<!ELEMENT aggregation (#PCDATA)>
<!ELEMENT update (attr, nexusdata)+>
<!ELEMENT attr (nexusdata)>
<!ATTLIST attr name NMTOKEN #REQUIRED>
```

Eine Anfrage in AWQL kann folgende Abschnitte enthalten. Der Föderation unterstützt die Abschnitte *aas*, *scope*, *restriction* und *filter.*

- AAS: Eine Liste von NAALs (*Nexus Augmented Area Locator*) gibt eine Menge von *Augmented Areas* (AAs) an, in denen bei dieser Anfrage gesucht werden soll. Eine Anwendung kann diesen Abschnitt dazu benutzen, um direkt bei einer Datenquelle zu suchen, damit die Föderation nicht nach den Spatial Model Servern suchen muß.
- SCOPE: Der *Scope*-Abschnitt gibt eine Liste mit *Extended Class Schema* (ECS) *indentifiers* an. Diese Schemas und das *Standard Class Schema* (SCS) definieren das Schema, das von der Anwendung, die eine Anfrage an die Föderation sendet benutzt und verstanden wird.
- RESTRICTION: Der *Restriction*-Abschnitt ist ein boolscher Ausdruck. Objekte werden nur dann zurückgegeben, wenn dieser Ausdruck für das Objekt wahr ist. In dem Ausdruck können mehrere boolsche Ausdrücke mit einem AND oder mit einem OR Element verknüpft werden. Mit dem NOT Element kann ein Wahrheitswert invertiert werden. Mit dem EQUAL Element wird überprüft, ob ein Objekt ein bestimmtes Attribut besitzt und ob es den angegebenen Wert hat. Ist dies der Fall, dann ist das EQUAL Element wahr für dieses Objekt. Mit dem IN Element wird überprüft, ob ein Objekt ein bestimmtes Attribut besitzt und ob es einen der angegebenen Werte hat. Das IN Element wird nur von AWQL unterstützt. AWQL 1.1 unterstützt das IN Element nicht mehr. Dafür unterstützt AWQL 1.1 zusätzlich zum EQUAL Element auch LIKE, GREATER und LESS Elemente. Ein GREATER oder LESS Elementen ist wahr, wenn der Wert eines Attributs größergleich bzw. kleiner als der Vergleichswert ist. Ein LIKE Element ist wahr für alle Objekte, deren durch den LESS Knoten vorgegebenes Attribut einen Wert besitzt, welcher durch das Pattern im Wert des LIKE Elements ausgedrückt werden kann. Die Elemente LIKE, GREATER und LESS werden von der zu entwerfenden Föderation für Vergleiche mit dem Attribut “type” nicht unterstützt. Das INSIDE Element ist für die Objekte wahr, wenn sicht das Gebiet des angegebenen Attributs in dem angegebenen geographischen Gebiet befinden. Das OVERLAPS Element ist für alle Objekte wahr, deren durch das Attribut angegebene Gebiet sich mit dem Gebiet des OVERLAPS Elements überschneidet. Dieses Gebiete sind in GML WKT kodiert (Siehe 3.1.6 WKT).
- FILTER: Durch den Filter-Abschnitt können die Attribute, die zurückgegeben werden eingeschränkt werden. Dies ist sinnvoll, wenn eine Anwendung nur an einem bestimmten Attribut eines Objekts interessiert ist.
- CLOSEST: Wird dieser Abschnitt angegeben, dann werden nur eine angegebene Anzahl von Objekten zurückgegeben, die einer ebenfalls angegebenen Position am nächsten sind. Diese Objekte müssen einen eventuell angegebenen RESTRICTION Abschnitt erfüllen. Die im Rahmen dieser Diplomarbeit implementierte Föderation unterstützt das CLOSEST Element (noch) nicht.

- GENERALIZATION: Geometrische Attribute können vereinfacht werden, d.h. kleine Details könne weggelassen werden. Dieser Abschnitt wird nicht von der Föderation unterstützt und ignoriert, sollte er in der Anfrage vorkommen.
- AGGREGATE: Mehrere kleine nebeneinanderliegenden Objekte können zu einem einzigen größeren Objekt zusammengefaßt werden. Dieser Abschnitt wird nicht von der Föderation unterstützt und ignoriert, sollte er in der Anfrage vorkommen.
- UPDATE: Mit diesem Abschnitt werden Attribute von bestimmten Objekten neue Werte zugewiesen. Update-Requests werden an die *Spatial Model Server* weitergereicht und von diesen ausgeführt.

3.1.2 AWML

Die von der Föderation zurückgegebenen Objekte werden mit der *Augmented World Modeling Language* repräsentiert. Auch die an die *insert* Methode der Föderation übergebenen Objekte und die von den *Spatial Model Servern* zurückgegebenen Objekte sind in AWML repräsentiert. Ein AWML Dokument kann mehrere Objekte repräsentieren.

Die DTD von AWML ist nicht mehr gültig. Sie wurde durch ein XML Schema ersetzt, ist hier jedoch angegeben, um einen Überblick über die Eigenschaften von AWML zu geben:

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<!ELEMENT awml (scope?, nexusobject)*>
<!ELEMENT scope (ecs)+>
<!ELEMENT ecs EMPTY>
<!ATTLIST ecs
    name CDATA #REQUIRED
    is NMTOKEN #IMPLIED>
<!ELEMENT nexusobject (#PCDATA)>
<!ATTLIST nexusobject
    type NMTOKEN #REQUIRED
    NOL CDATA #REQUIRED
    kind(virtual | real) “real”>
```

Die innerhalb des *nexusobject* Elements repräsentierten Objekte müssen gültige Objekte in der *Augmented World* sein. Zum Parsen und Erzeugen werden die Klassen *ResultSetFactory* und *ResultSetWriter* verwendet. Da diese Klassen das *scope* Element nicht unterstützen, unterstützt es die Föderation auch (noch) nicht. Es ist sinnvoll diese Klassen dementsprechend zu erweitern, damit das *scope* Element unterstützt wird.

Mit dem *scope* Abschnitt kann das für diese Objekte gültige Schema angegeben werden. Durch das Schema wird unter anderem angegeben, welche Attribute ein Objekt besitzen muß und welche es besitzen kann aber nicht muß.

Beispiel eines AWML Dokuments das an *globalInsert* übergeben wird:

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<awml>
    <nexusobject type=”plane” NOL=”nexus:http://airport.galway.ie:8080/soap/servlet/
rpcrouter|urn:QueryComponent|0x00000000000000000000000000000001/
0x34567890123456789012345678901234”>
        <name>EW 4684</name>
        <speed>0</speed>
        <maxspeed>278</maxspeed>
        <pos>
            <WKT>POINT Z(52.7011 -8.92083 0)</WKT>
        </pos>
```

```
        <accuracy>5</accuracy>
    </nexusobject>
</awml>
```

3.1.3 Der Nexus Object Locator (NOL)

Der *Nexus Object Locator* repräsentiert ein Referenz auf ein Nexus Objekt.

NOLs bzw. NELs (Nexus Entity Locators) haben folgendes Format:

```
"nexus:"<schemaid>":"<schemspecstr>"|"(<targetobjecturi>)?("|"<aaid>("/
"<objectid>)?)?
```

Also zum Beispiel:

```
nexus:http://airport.galway.ie:8080/soap/servlet/rpcrouter|urn:QueryCompo-
nent|0x00000000000000000000000000000001/0x3456789012345678901234567890123 4
```

Der Abschnitt zwischen “nexus:” und dem ersten “|” (also <schemaid>:<schemspecstr>) wird als URL für einen SOAP Aufruf benutzt. Der Abschnitt zwischen dem ersten “|” und dem zweiten “|” (also <targetobjecturi>) wird dazu benutzt um den SOAP Service anzugeben, an den eine Anfrage gestellt werden soll. Mit <aaid> und <objectid> werden die *AugmentedArea ID* und die *Object ID* angegeben. Sie sind die Stringrepräsentation einer UUID (32 hex characters, also z.B. “0x123515...”)

Wenn <schemaid> “http” ist, dann hat <schemaspectr> diese Struktur:

```
"//"<servername>(":"<port>)?("/"<appendix>)?
```

Also zum Beispiel:

```
airport.galway.ie:8080/soap/servlet/rpcrouter
```

Port und der Pfad *appendix* sind optional. Werden sie nicht angegeben, dann werden folgende Default-Werte verwendet:

```
port = 8080

appendix = soap/servlet/rpcrouter

targetobjecturi = urn:QueryComponent
```

3.1.4 AADL

Um Anfragen an das *Area Service Register* zu stellen, übergibt die Föderation dem ASR ein AADL Dokument. Das ASR gibt in seiner Antwort eine Anzahl von AADL Dokumenten in einem AAList Dokument zurück.

Die DTD von AADL ist hier angegeben, um einen Überblick über die Eigenschaften von AADL zu geben:

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<!ELEMENT aadl (resultspec?, naal?, area?, awschema?) >
<!ELEMENT resultspec EMPTY>
<!ATTLIST resultspec
      naal     (yes|no) “yes”
      area     (yes|no) “no”
      awschema (yes|no) “no”>
```

```
    <!-- all stored information on objects (objclass,lod,...) -->
<!ELEMENT naal (#PCDATA) >
<!ELEMENT area (gml)>
    <!-- gml is defined in another DTD -->
<!ELEMENT awschema (objectclass)+ >
<!ELEMENT objectclass EMPTY>
<!ATTLIST objectclass
      type   NMTOKEN #IMPLIED
      lod    NMTOKEN #IMPLIED
      lodmin NMTOKEN #IMPLIED
      lodmax NMTOKEN #IMPLIED>
```

Mit dem *area* Element wird dem *Area Service Register* angegeben, in welchen geographischen Regionen Nexus Objekte gesucht werden. Dabei wird ein Polygon oder Multi-Polygon angegeben.

Mit dem *awschema* Element wird dem *Area Service Register* angegeben, welche Nexus Klassen von zu findenden Nexus Servern zurückgegeben werden sollen können.

In Zukunft soll es möglich sein, bestimmte Eigenschaften von Nexus Servern der Diensteschicht beim Area Service Register anzugeben und abzufragen. Damit kann zum Beispiel angegeben werden, ob ein Server in der Lage ist einen *restriction* Ausdruck zu parsen.

Dies entspricht dieser Änderung im DTD der AADL:

```
<!ATTLIST resultspec
      naal     (yes|no)    “yes”
      area     (yes|no)    “no”
      awschema (yes|no)    “no”
      capabilities (yes|no) “no”>
<!ELEMENT capabilities EMPTY>
<!ATTLIST capabilities sorted (true|false) #REQUIRED
                       ignoresRestriction (true|false) #REQUIRED>
```

Werden AADL Dokumente vom ASR zurückgegeben, dann wird mit dem *area* Element, dem *awschema* Element und dem *capabilities* Element das Dienstgebiet des Servers, die Nexus Objekte welche von diesem Server zurückgegeben werden können und die Eigenschaften des Servers der Föderation übergeben.

3.1.5 CRL

Die Funktionen *globalUpdate*, *globalInsert* und *globalDelete* der Föderation geben ein XML Dokument in CRL zurück. In einem solchen Dokument wird für jedes Objekt, für das eine Operation ausgeführt werden sollte zurückgemeldet, ob die Operation erfolgreich war.

Die DTD von CRL ist hier angegeben, um einen Überblick über die Eigenschaften von CRL zu geben:

```
<xml version=”1.0” encoding=”US-ASCII”?>
<!ELEMENT changereport (action)*>
<!ELEMENT action #PCDATA>
<!ATTLIST action
    type #PCDATA      #REQUIRED
    obj   #PCDATA      #REQUIRED
    res (success | failed) #REQUIRED>
```

Antwort in CRL der erfolgreichen Registrierung des mobilen Objektes des letzten Beispiels:

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<changereport>
      <action  type=”insert”  obj=”nexus:http://airport.galway.ie:8080/soap/servlet/
rpcrouter|urn:QueryComponent||0x00000000000000000000000000000001/
0x34567890123456789012345678901234” res=”success”/>
</changereport>
```

In jedem *action* Element wird eine NOL mit dem Attribut *obj* zurückgegeben. Sie dient der Zuordnung der jeweiligen Antworten in den *action* Elementen zu den jeweiligen *nexusobject* Elementen in der Anfrage in AWML (*insert*) oder um der Anwendung mitzuteilen, für welche Nexus Objekte versucht wurde Attribute zu ändern oder für welche Nexus Objekte versucht wurde diese zu entfernen.

3.1.6 WKT

Koordinaten oder Polygone, die der Föderation übergeben werden oder von der Föderation zurückgegeben werden sind in der WKT-Notation repräsentiert. Hier ist die Spezifikation aus [IBMDB2] wiedergegeben:

```
<Geometry Tagged Text> := <Point Tagged Text> | <LineString Tagged Text>
                                              | <Polygon Tagged Text>
                                              | <MultiPoint Tagged  Text>
                                              | <MultiLineString Tagged Text>
                                               | <MultiPolygon Tagged Text>

<Point Tagged Text> := POINT (<Point Text>)

<LineString Tagged Text> := LINESTRING (<LineString Text>)

<Polygon Tagged Text> := POLYGON (<Polygon Text>)

<MultiPoint Tagged Text> := MULTIPOINT (<MultiPoint Text>)

<MultiLineString Tagged Text> := MULTILINESTRING (<MultiLineString Text>)

<MultiPolygon Tagged Text> := MULTIPOLYGON (<MultiPolygon Text>)

<Point Text> := EMPTY |    <Point> | Z  (<PointZ>) | M  (<PointM>) | ZM (<PointZM>)

<Point> :=  <x>  <y>
<x> := double precision literal
<y> := double precision literal

<PointZ> :=  <x>  <y>  <z>
<x> := double precision literal
<y> := double precision literal
<z> := double precision literal

<PointM> :=  <x>  <y>  <m>
<x> := double precision literal
<y> := double precision literal
<m> := double precision literal

<PointZM> :=  <x>  <y>  <z>  <m>
<x> := double precision literal
<y> := double precision literal
<z> := double precision literal
<m> := double precision literal

<LineString Text> := EMPTY
                                    |  ( <Point Text >   {,  <Point Text> }*  )
                                    | Z  ( <PointZ Text >  {,  <PointZ Text> }*  )
                                    | M  ( <PointM Text >  {,  <PointM Text> }*  )
```

```
                                        | ZM ( <PointZM Text > {,  <PointZM Text> }*  )

<Polygon Text> := EMPTY | ( <LineString Text > {,< LineString Text > }*  )

<MultiPoint Text> := EMPTY | ( <Point Text >   {,  <Point Text > }*  )

<MultiLineString Text> := EMPTY | ( <LineString Text > {,< LineString Text>}*  )

<MultiPolygon Text> := EMPTY | ( < Polygon Text > {,  < Polygon Text > }*  )
```

Ein Punkt kann in Nexus durch einer dieser Strings repräsentiert werden:

```
<wkt srsCode=”4326”>POINT (<x> <y>)</wkt>
<wkt>POINT (<y> <x>)</wkt>
```

Ein Polygon kann in Nexus durch einer dieser Strings repräsentiert werden:

```
<wkt srsCode=”4326”>POLYGON  ((<x> <y>,<x> <y>,...,<x> <y>))</wkt>
<wkt>POLYGON  ((<y> <x>,<y> <x>,...,<y> <x>))</wkt>
```

Die Koordinaten können in WGS84, Gauss Krüger Zone 3+4 und UTM Zone 32N angegeben werden. Bei Nexus ist die Reihenfolge der <x> und der <y> Koordinaten im Gegensatz zu der WKT Spezifikation (Siehe “[IBMDB2]”) allerdings vertauscht, wenn kein Koordinaten-System mit dem Attribut *srsCode* angegeben wird.

Bei WGS84 ist <x> der Breitengrad, <y> der Längengrad und <z> die relative Höhe einer Position in Metern.

3.2 Die Java-Klassen

Es werden nur die wichtigsten Nexus Java-Klassen vorgestellt. Klassen die für die Funktion der Föderation zweitrangig sind werden nicht vorgestellt.

3.2.1 Die Geoklasse CGeometry

Die abstrakte Klasse *CGeometry* und ihre Kind-Klassen befinden sind im Package *de.uni_stuttgart.nexus.geoClasses*. Die Schnittstellen der Geo-Klassen sind in [OpenGIS] näher beschrieben.

Die Klasse *CGeometry* implementiert die Methoden *union_op* und *intersection*. Diese Methoden geben die Union bzw. die Intersection mit dem übergebenen *CGeometry* Objekt zurück. Nachdem der *Restriction*-Ausdruck in eine DNF konvertiert wurde werden die Methoden benötigt, um Polygone innerhalb einer Konjunktion zu einem *CGeometry* Objekt zu vereinigen, falls möglich. Sie werden ebenfalls dazu verwendet, um Konjunktionen semantisch miteinander zu vereinigen, falls dies möglich ist.

Soll der *Restriction*-Ausdruck von der Föderation geparst werden wenn ein Nexus Server der Diensteschicht dies nicht unterstützt, dann werden die boolschen Bedingungen der Knoten, wie zum Beispiel der *overlaps* und *inside* Elemente für jedes von dem Nexus Server erhaltene Nexus Objekt durch die Klasse *ResultSet* selbst überprüft. Diese Klasse verwendet dazu die Methoden *intersects* (Intersection Matrix T********) für *overlaps* Elemente und *within* (Intersection Matrix T*F**F***) für *inside* Elemente.

Durch eine Intersection Matrix [ArcSDE] wird angegeben, in welcher Beziehung zwei Geometrischen Objekte zueinander stehen müssen, damit die boolsche Bedingung der Operation

wahr ist.

Tabelle 1: Die Intersection Matrix von intersects

	Interior (a)	Boundary(a)	Exterior(a)
Interior(b)	T	*	*
Boundary(b)	*	*	*
Exterior(b)	*	*	*

Die Einträge in der Matrix haben folgende Bedeutung:

T: Die entsprechenden Gebiete der Geographischen Objekte müssen sich schneiden. Die Dimension (0-2) ist beliebig.
F: Die entsprechenden Gebiete der Geographischen Objekte dürfen sich nicht schneiden (Dimension -1).
*****: Es ist egal, ob sich die entsprechenden Gebiete scheiden oder nicht:
0,1 oder 2: Die entsprechenden Gebiete müssen sich schneiden und die Dimension darf höchstens 0,1 oder 2 sein.

Die Intersection Matrix wird von Links-Oben nach Rechts-Unten gelesen.

3.2.2 Die AWS Klassen

Das Klassen im Package *de.uni_stuttgart.nexus.shared.aws* werden zur Repräsentation und zum Parsen eines Schemas verwendet, wie das *Augmented World Schema* oder die *Extended Class Schemata.* Eine detaillierte Dokumentation dieser Klassen findet sich in [Enge 2003].

3.2.2.1 AWSFactory

Die Klasse *AWSFactory* liest ein *AWS* Dokument ein und erzeugt ein *AWS* Objekt. Der Methode *createAWS* wird ein *Reader* Objekt übergeben. Die Methode gibt ein *AWS* Objekt zurück.

3.2.2.2 AWS

Die Klasse repräsentiert ein geparstes *AWS* Dokument. In einem *AWS* Objekt werden die *NexusAttribute* Objekte, die *NexusDataType* Objekte und die *NexusType* Objekte eines Schemas gespeichert. Die Objekte sind in *TreeMaps* gespeichert. Die Klasse bietet Methoden an, um solche Objekte dem *AWS* Objekt hinzuzufügen oder um sie auszulesen.

3.2.2.3 NexusAttribute

Ein *NexusAttribute* Objekt repräsentiert einen Attribut im *AWS*, wie zum Beispiel *Position* oder *NOL*. Für jedes im *AWS* gefundene Attribut mit dem selben Namen gibt es nur ein *NexusAttribute* Objekt. Diese Objekt besitzt neben dem Namen des Attributs eine ID, durch welche es eindeutig identifiziert ist. Desweiteren besitzt es ein *NexusDataType*, welches den Datentyp des Attributs bestimmt.

3.2.2.4 NexusDataType

Ein NexusDataType Objekt repräsentiert den Typ eines Attributs, wie zum Beispiel *CGeometry*, *String* oder *NOL*. Die Objekte dieser Klasse geben an, welches Syntax das Attribut besitzen muß, um gültig zu sein und wie Daten verglichen werden.

3.2.2.5 NexusType

Ein NexusType Objekt repräsentiert einen Typ, den ein Nexus Objekt haben kann, also die Klasse des Nexus Objekts. Jedes NexusType Objekt besitzt eine Referenz auf seinen Supertype und auf alle Sub-Typen, sowie ein Liste (eine TreeMap) mit den optionalen Attributen und mit den notwendigen Attributen. Diese Attribute sind durch Objekte der Klasse *NexusAttribute* Repräsentiert.

3.2.3 Die AWQL Klassen

Die Klassen werden dazu verwendet, um ein *AWQL* Dokument zu repräsentieren und um auf es zuzugreifen.

3.2.3.1 AWQLQuery

Ein Objekt dieser Klasse repräsentiert ein *AWQL* Dokument. Der *restriction*-Ausdruck wird durch *RestrictionOperator* Objekte bzw. dessen Sub-Klassen repräsentiert. Der CLOSEST Ausdruck wird durch ein *NearestNeighborOperator* Objekt repräsentiert. Der AAS Ausdruck wird durch ein *AAList* Objekt repräsentiert. Der SCOPE Ausdruck wird durch ein *ECStoAlias-Mapping* Objekt repräsentiert. Der FILTER Ausdruck wird durch ein *AttributeFilter* Objekt repräsentiert und der UPDATE Ausdruck wird durch ein *AVList* Objekt repräsentiert.

3.2.3.2 AWQLQueryWriter

Die *AWQLQueryWriter* Klasse konvertiert ein *AWQLQuery* Objekt in ein *AWQL* Dokument.

3.2.3.3 AWQLQueryFactory

Die *AWQLQueryFactory* Klasse konvertiert ein *AWQL* Dokument in ein *AWQLQuery* Objekt.

3.2.4 Die AWML Klassen

Das Klassen im Package *de.uni_stuttgart.nexus.shared.awml* werden zur Repräsentation von Ergebnismengen und zum Arbeiten auf solchen Ergebnismengen verwendet. Die Klassen *ResultSetFactory* und *ResultSetWriter* werden dazu verwendet, um ein AWML Dokument in ein *ResultSet* Objekt zu konvertieren oder um ein *ResultSet* Objekt in ein AWML Dokument zu konvertieren. Eine detaillierte Dokumentation dieser Klassen findet sich in [Enge 2003].

3.2.4.1 GenericObject

In dieser Klasse werden die Daten eines Nexus Objekts gespeichert. Ein Nexus Objekt kann mehrere Typen und *NOLs* besitzen. Das Objekt besitzt Methoden um *NexusAttribute* Objekte und einen dazugehörigen Wert hinzuzufügen und um diese auszulesen. Jedes Attribut kann mehr als einen Wert besitzen.

3.2.4.2 ResultSet

Ein *ResultSet* repräsentiert eine Ergebnismenge. In einem *ResultSet* werden *GenericObject* Objekte gespeichert. Einem *ResultSet* können für Attribute Indizes auf die Werte von dem jeweiligen Attribut hinzugefügt werden. Dadurch können Objekte, dessen Attribut einen bestimmten Wert hat bei dem Aufruf der *query* Methode effizienter gefunden werden.

Das *ResultSet* Objekt implementiert eine *query* Methode. Der Methode wird ein *ResultSetFilter* Objekt übergeben. Die Methode gibt in einem *ResultSet* die *GenericObject* Objekte zurück, welche die boolsche Bedingung des *ResultSetFilter* Objekts erfüllen.

3.2.4.3 ResultSetFactory

Die *ResultSetFactory* Klasse liest ein *AWML* Dokument und konvertiert es in ein *ResultSet* Objekt.

3.2.4.4 ResultSetFilter

Ein ResultSetFilter Objekt speichert eine Anzahl von boolschen Bedingungen, welche von *GenericObjects* erfüllt werden sollen. Ein ResultSetFilter Objekt kann eine Liste von oder-verknüpften Nexus Typen speichern. Ebenfalls kann in einem ResultSetFilter Objekt ein CGeometry Vergleichs-Area und die Namen von den Attributen für die der Vergleich durchgeführt werden soll gespeichert werden. Um beliebige andere Attribute, wie zum Beispiel das NOL Attribut zu überprüfen, können *ResultSetFilterAttribute* Objekte im ResultSetFilter gespeichert werden.

3.2.4.5 ResultSetFilterPredicate

Ein *ResultSetFilterPredicate* enthält ein *NexusAttribute* Objekt und ein oder mehrere Vergleichswerte sowie die Art des Vergleichs, wie zum Beispiel *equal*.

3.2.4.6 ResultSetMerger

Die *ResultSetMerger* Klasse vereinigt zwei *ResultSets* in ein *ResultSet*. Kommt ein *GenericObject* in beiden *ResultSets* vor, dann werden beide Objekte vereinigt. Dabei werden die Attribute beider Objekte in dem vereinigten Objekt gespeichert. Ein Attribut kann also mehr als einmal vorkommen.

3.2.4.7 ResultSetWriter

Die *ResultSetWriter* Klasse konvertiert ein *ResultSet* in ein *AWML* Dokument.

3.2.5 Klasse zur Kommunikation mit Servern: ContactServer

Die Klasse *ContactServer* übergibt Requests an *Spatial Model Server* und den *AWQLWrapper* und gibt das Ergebnis zurück. Den Methoden werden *AWQLQuery* und *ResultSet* Objekte übergeben. Es werden *CRLSet* und *ResultSet* Objekte zurückgegeben.

3.2.6 CRL Klassen

3.2.6.1 CRLSet

Die Klasse wird dazu verwendet, um *CRL* Dokumente zu repräsentieren.

3.2.6.2 CRLResult

Die Klasse wird repräsentiert jeweils einen Eintrag eines *CRL* Dokuments.

3.3 Apache Tomcat und Apache SOAP

Auf die Funktionen der Föderation, also das *Nexus Application Interface*, auf das ASR und auf die Nexus Server der Diensteschicht, also *Nexus Service Interfaces*, soll mit SOAP zugegriffen werden können (Siehe “[Apache SOAP]”).

Die Schnittstelle der *Föderation wird in einer Klasse Federation implementiert. Sie besitzt die Methoden query, globalUpdate*, *globalInsert* und *globalDelete*. Diese Klasse wird von Apache Tomcat gehostet und der SOAP remote procedure call (RPC) aufgerufen. Bei dem Aufruft einer remote procedure wird die entsprechende Methode der Klasse *Federation* aufgerufen.

Um auf die Föderation per SOAP RPC zugreifen zu können, wird Apache SOAP benutzt. Apache SOAP bietet eine Library für die Clientseite an, die es ermöglicht *remote procedure call requests* per SOAP zu verschicken und das empfangene XML Dokument zu decodieren und als Rückgabe des RPC zurückzugeben. Apache SOAP stellt auch ein RPC Servlet für die Serverseite zu Verfügung, das die entsprechenden Methoden eines Objekts, auf die per RPC zugegriffen werden soll aufruft und das Ergebnis in ein XML Dokument codiert und zurücksendet.

Dieses Servlet muß von einem Servlet-Container gehostet werden, wie zum Beispiel IBM Web Sphere oder Apache Tomcat. Die Aufgabe des Servlet-Containers ist es, die per HTTP gestellten Anfragen entgegenzunehmen und an das richtige Servlet weiterzureichen, sowie das Ergebnis über die selbe Verbindung zurückzugeben. Um das Apache SOAP Servlet *rpcrouter* zu hosten, wird der Servlet-Container Apache Tomcat benutzt

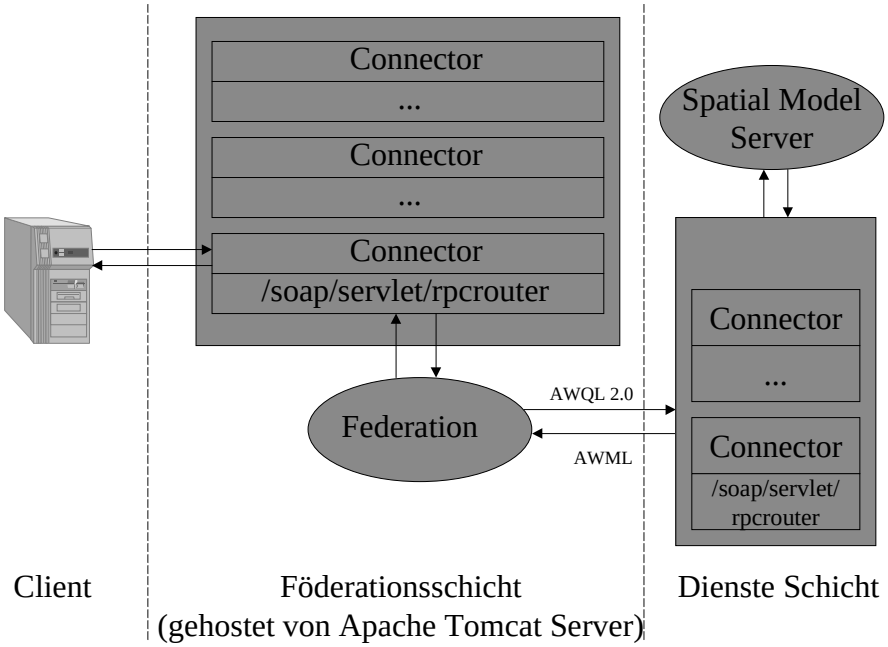


Abb. 5: Der Servlet Container

Auf einem Apache Tomcat Server können ein oder mehrere Service Komponenten installiert werden. Jede Service Komponente benötigt zumindest einen Connector und genau eine Engine. Ein Connector wartet auf einem bestimmten Port auf TCP Verbindungen, um Requests per HTTP entgegen zu nehmen. Eine Service Komponente hat nur eine Engine. Diese muß die Anfragen von allen Connector Komponenten ihres Services weiterleiten. Eine Engine verwaltet eine oder mehrere Host Komponenten. Eine Host Komponente repräsentiert den Namen des Rechners, der im DNS bekannt sein sollte. Die Host Komponente verwaltet einen oder mehrere Context Komponenten. Eine Context Komponente verwaltet eine Web Anwendung, wie zum Beispiel Apache SOAP und besitzt einen Kontextpfad. Der Kontextpfad von Apache SOAP ist */soap*.

Um einen empfangenen Request an die richtige Web Anwendung weiterzuleiten, vergleicht Tomcat den Anfang des Pfades des Requests (zum Beispiel */soap/servlet/rpcrouter*) mit den Kontextpfaden der Kontext Komponenten, und gibt den Request an die Web Anwendung der Kontext Komponente mit der längsten Übereinstimmung weiter. Die Apache SOAP Komponente wählt das aufzurufende Servlet (zum Beispiel rpcrouter) anhand des Servlet mappings des Deployment Descriptors der Web Anwendung.

Das rpcrouter Servlet wählt das Objekt und die Methode, die aufgerufen werden soll anhand der Deployment Descriptoren aller beim rpcrouter registrierten Objekte aus.

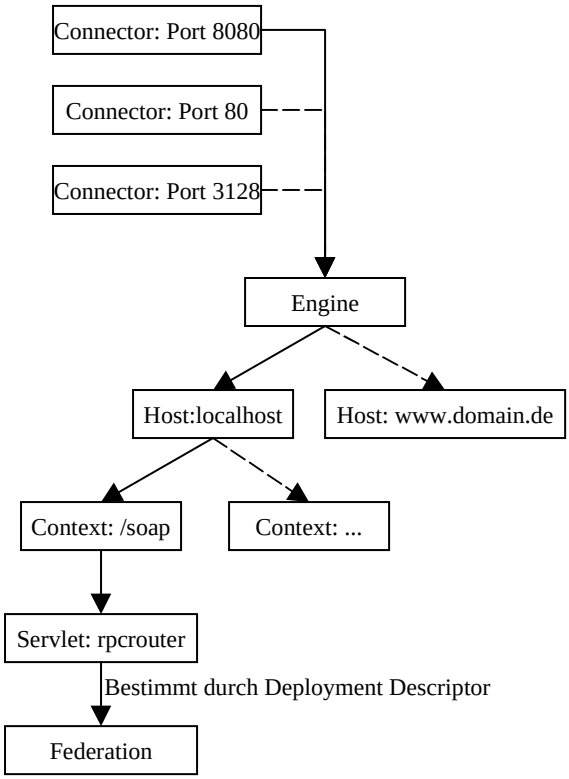


Abb. 6: Anfrageweiterleitung innerhalb des Apache Tomcat

Ein Deployment Descriptor ist ein XML Dokument. Das zu entwerfende Objekt wird mit diesem Befehl registriert:

```
java org.apache.soap.server.ServiceManagerClient
     http://<host>:<port>/soap/servlet/rpcrouter deploy <deploymentDescriptor>
```

<host> ist der Hostname oder die IP Adresse, auf der Apache Tomcat gehostet wird.
<port> ist der Port, an den Requests für das rpcrouter Servlet gesendet werden müssen.
<deploymentDescriptor> ist der Dateiname der DeploymentDescriptor Datei.

Bei Nexus müssen die Requests an Port 8080 gesendet werden.

Damit Apache SOAP Objekte, die per SOAP übergeben werden sollen in XML kodieren kann, wird für jede zu übergebende Klasse ein sogenannter Serializer benötigt. Er konvertiert die Attribute eines Java-Objekts in XML Code. Um aus einem XML Dokument ein Java-Objekt zu erzeugen, wird ein sogenannter Deserializer benötigt. Für die Standardtypen definiert SOAP selbst, wie diese zu serialisieren sind. Für diese Standardtypen stellt Apache SOAP Serializer und Deserializer zur Verfügung.

Für Apache SOAP unbekannte Klassen müssen deren Serializer und Deserializer bei dem rpcrouter Servlet registriert werden. Von den Komponenten der Föderation werden allerdings nur String Objekte übergeben und die Föderation erhält nur String Objekte. Es müssen deshalb keine eigenen Serializer oder Deserializer registriert werden.

Für die Registrierung der Föderation wird dieser Deployment Descriptor verwendet:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QueryComponent">
  <isd:provider type="java"
                scope="Application"
        methods="query insert update delete">
    <isd:java class="de.uni_stuttgart.nexus.federation.Federation" static="false"/>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>
  <isd:mappings/>
</isd:service>
```

3.4 Anbindung des Cache

Anfragen welche an die Föderation gestellt werden können sich auf Gebiete beziehen, welche bereits Gegenstand einer vorherigen Anfrage waren. Dabei könnte ein Teil der Anfrage mit dem Resultat der vorherigen Anfrage beantwortet werden. Ein solcher Cache ist bei der bisherigen Föderation nicht implementiert, wird aber in einer parallelen Diplomarbeit entwickelt. Ein solcher Cache sollte dazu in der Lage sein:

1. Eine Menge von Anfragen soll an den Cache übergeben werden können. Der Cache soll eine neue Menge von Anfragen zurückgeben, welche äquivalent zur übergebenen Menge von Anfragen ist und welche idealerweise Anfragen an den Cache enthält.
2. Eine *query* Anfrage soll an den Cache gestellt werden können. Der Cache soll diese Anfrage in *AWQL* genau so wie ein Nexus Server der Dienste Schicht verarbeiten und Nexus Objekte zurückgeben.
3. Objekte sollen in den Cache eingefügt werden können. Da der Cache den Kontext kennen muß, unter dem diese Objekte eingefügt werden sollen, muß die ursprüngliche *AWQL* Anfrage mit der diese Objekte von einem Nexus Server der Dienste Schicht angefordert wurden mit übergeben werden. Das ist deshalb notwendig, da der Cache zum Beispiel wissen muß, welche Attribute wegen eines *filter* Elements in der Anfrage möglicherweise nicht zurückgegeben wurden und welche boolschen Bedingungen im *restriction* und *closest* Element diese Objekte erfüllen mußten.

4. Objekte sollen im Cache aktualisiert werden können, wenn die Föderation ein Update durchführt. Dazu wird dem Cache ein *AWQL* Request übergeben. Dadurch wird sichergestellt, daß die Föderation keine veralteten Nexus Objekte aus dem Cache zurückgibt, obwohl diese Objekte bereits durch die Föderation aktualisiert wurden. Dabei können allerdings veraltete Nexus Objekte zurückgegeben werden, wenn diese direkt beim Nexus Server und nicht über die aktuelle Föderation aktualisiert wurden.
5. Objekte sollen aus dem Cache entfernt werden können. Dazu wird dem Cache ein *AWQL* Request übergeben.

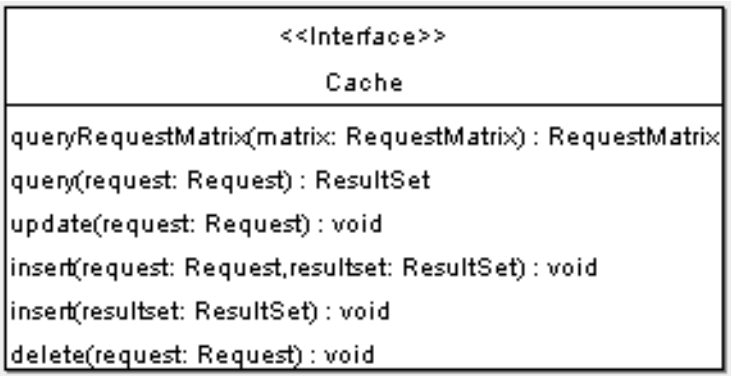


Abb. 7: Schnittstelle des Cache

Der Aufbau der Klassen *Request* und *RequestMatrix* ist in [6.7 Sonstige Klassen] genauer beschrieben. Ein *Request* Objekt enthält sowohl ein *AWQLQuery* als auch ein *Server* Objekt. Sowohl das *AWQLQuery* Objekt als auch das *Server* Objekt enthalten ein *RestrictionOperator* Objekt. Das *RestrictionOperator* Objekt des *AWQLQuery* Objekts ist der Wurzelknoten des an die Föderation übergebenen Restriction Ausdrucks. Die *Föderation* konvertiert diesen Ausdruck in eine *DNF* und baut für jeden anzufragenden *Spatial Model Server* einen *Restriction* Ausdruck auf. Dieser zu verwendende *Restriction* Ausdruck wird im *Server* Objekt abgelegt. Die Referenz in *AWQLQuery* referenziert keinen gültigen *Restriction* Ausdruck mehr und darf nicht vom Cache verwendet werden.
Bei der Rückgabe eines neuen Requests ist die *restriction* im *RestrictionOperator* Objekt des *Server* Objekts zurückzugeben. Dieser *RestrictionOperator* muß ein *restriction*-Ausdruck in *DNF* repräsentieren, wenn der Server an den der Request übergeben werden soll nicht dazu in der Lage ist, den *restriction* Ausdruck zu verarbeiten, also das *ignoresRestriction* Flag in seinem *Capabilities* Objekt gesetzt ist, da der *restriction*-Ausdruck in diesem Fall von der Föderation auf die zurückgegebenen Objekte (die *ResultSets*) angewendet werden muß und der anwendende Operator davon ausgeht, daß der *restriction*-Ausdruck in *DNF* ist.

4 Architektur

4.1 Die Schnittstelle der Föderation

Die Föderation besitzt folgende Methoden, an welche Anfragen gestellt werden können:

query (in: AWQLQuery out: ResultSet)

Der *query* Methode wird ein *AWQLQuery* Objekt übergeben. Die Methode gibt ein *ResultSet* zurück.

query (in: String out: String)

Der *query* Methode wird ein *AWQL* Dokument übergeben. Die Methode konvertiert das *AWQL* Dokument in ein *AWQLQuery* Objekt, ruft die obige *query* Methode auf, konvertiert das *ResultSet* in ein *AWML* Dokument und gibt dies zurück.

update (in: AWQLQuery out: CRLSet)

Der *update* Methode wird ein *AWQLQuery* Objekt übergeben. Die Methode gibt ein *CRLSet* Objekt zurück.

update (in: String out: String)

Der *update* Methode wird ein *AWQL* Dokument übergeben. Die Methode konvertiert das *AWQL* Dokument in ein *AWQLQuery* Objekt, ruft die obige *update* Methode auf, konvertiert das *CRLSet* Objekt in ein *CRL* Dokument und gibt dies zurück.

insert (in: ResultSet out: CRLSet)

Der *insert* Methode wird ein *ResultSet* Objekt übergeben. Die Methode gibt ein *CRLSet* Objekt zurück.

insert (in: String out: String)

Der *insert* Methode wird ein *AWML* Dokument übergeben. Die Methode konvertiert das *AWML* Dokument in ein *ResultSet* Objekt, ruft die obige *insert* Methode auf, konvertiert das *CRLSet* Objekt in ein *CRL* Dokument und gibt dies zurück.

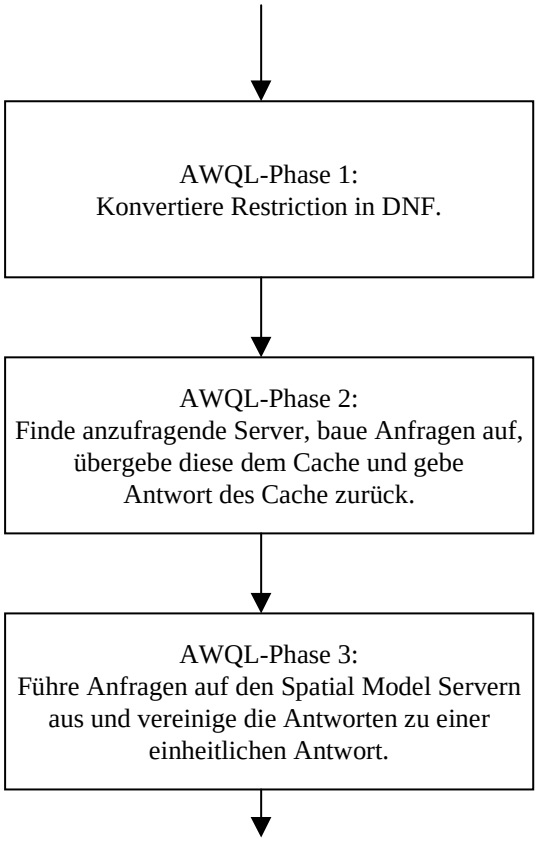
delete (in: AWQLQuery out: CRLSet)

Der *delete* Methode wird ein *AWQLQuery* Objekt übergeben. Die Methode gibt ein *CRL*Set Objekt zurück.

delete (in: String out: String)

Der *delete* Methode wird ein *AWQL* Dokument übergeben. Die Methode konvertiert das *AWQL* Dokument in ein *AWQLQuery* Objekt, ruft die obige *delete* Methode auf, konvertiert das *CRLSet* Objekt in ein *CRL* Dokument und gibt dies zurück.

4.2 Ausführen von query



Die *query* Methode führt die drei *AWQL*-Phasen aus. Die Algorithmen der Phasen sind jeweils in Operatoren implementiert. Die *query* Methode baut den Operatoren-Baum der jeweiligen Phase auf und führt eine Anfrage auf dem jeweiligen Anfrage-Baum aus.

Mit dem *AWQL* Dokument wird ein Dokument übergeben, welches für Nexus Objekte eine boolsche Bedingung darstellt. Die Anfrage soll nur für die Objekte ausgeführt werden, welche diese boolsche Bedingung erfüllen. Sollen Nexus Objekte zurückgegeben werden, dann kann in dem *AWQL* Dokument angegeben werden, welche Attribute zurückgegeben werden sollen. Sollen dagegen Attribute aktualisiert werden, dann werden auch diese in dem *AWQL* Dokument mit übergeben.

Um bestimmen zu können, welche Nexus Objekte die boolsche Bedingung eines *AWQL* Dokuments erfüllen, muß das *AWQL* Dokument analysiert werden und es muß bestimmt werden, auf welchen Servern die Objekte gespeichert sind.

In einem *AWQL* Dokument wird die boolsche Bedingung mit dem *restriction* und dem *closest* Element übergeben. In dieser Diplomarbeit werden nur die Fälle behandelt, in denen kein *closest* Element übergeben wird.

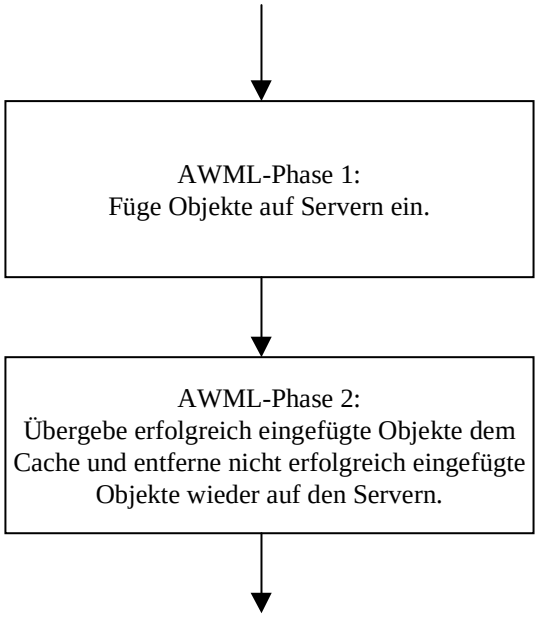
Es muß also die boolsche Bedingung unterhalb des *restriction* Elements ausgewertet werden. Um zu bestimmen, an welche Server Anfragen gestellt werden sollen, werden Anfragen an das *ASR* gestellt. Um die notwendigen Anfragen an das *ASR* verringern zu können und die Anfragen an die Server möglichst parallel und voneinander unabhängig ausführen zu können, soll der boolsche Ausdruck des *restriction* Ausdrucks in einer ersten Phase in eine distributive Normalform konvertiert werden. Diese Normalform kann dann in dieser ersten Phase noch etwas

zusammengefaßt werden, indem auch die Semantik der Prädikate berücksichtigt wird.

In einer zweiten Phase wird unter Verwendung des *ASR* festgestellt, an welche Server Anfragen gestellt werden müssen. Wurden im *AWQL* Dokument diese Server bereits angegeben, dann ist eine Anfrage an das *ASR* um die Server zu finden nicht mehr notwendig. Aus den gefundenen Servern und den jeweiligen Klauseln der *DNF*, welche an diesen Server in einer Anfrage an diesen übergeben werden sollen, wird ein *RequestMatrix* Objekt erstellt. Es repräsentiert alle Anfragen, welche an die Server zu stellen sind. Dieses Objekt wird dem Cache übergeben. Der Cache gibt ein *RequestMatrix* Objekt zurück. In dem zurückgegebenen *RequestMatrix* Objekt kann auch der Cache als Server vorkommen, an den eine Anfrage gestellt werden soll.

In der dritten Phase sollen die Anfragen an die Server parallel gestellt werden und die Antworten zusammengefaßt werden.

4.3 Ausführen von insert



Die *insert* Methode führt drei Phasen durch. Die einzelnen Phasen werden mit Operatoren implementiert. Die *insert* Methode baut den Operatoren-Baum der jeweiligen Phase auf und führt eine Anfrage auf dem jeweiligen Anfrage-Baum aus.

Wenn der Methode *insert* ein *AWML* Dokument übergeben wird, dann müssen die übergebenen Nexus Objekte an die *insert* Methoden der Server übergeben werden.

Vor AWML-Phase 1 sortiert daher die *insert* Methode die Nexus Objekte welche der *insert* Methode übergeben wurden nach Zielservern in *ResultSets*. Diese Server werden durch die *SpaSeLocator* der *NOLs* der Objekte bestimmt.

Besitzt ein Objekt keine *NOL*, dann muß eine erzeugt werden. Der *SpaSeLocator* des Objekts wird dadurch gefunden, indem ein Server für das Objekt gefunden wird. Dazu wird eine Anfrage an das *ASR* gestellt mit dem Typ bzw. den Typen des Objekts und einem Gebiet in dem sich das Objekt befindet. Für jeden vom *ASR* gefundenen Server wird eine *NOL* erzeugt mit der selben *ObjectId*. Diese *NOLs* werden dem Objekt hinzugefügt.

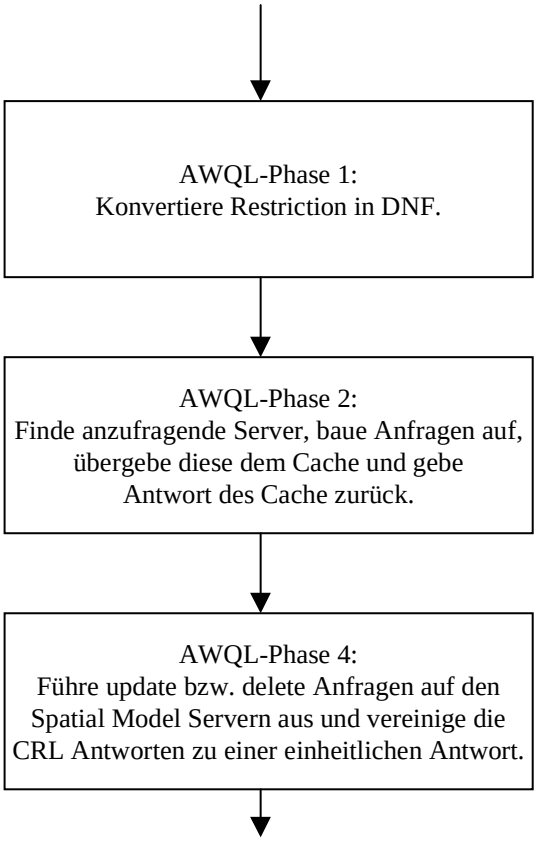
Nachdem die Objekte nach Servern sortiert wurden, wird AWML-Phase 1 ausgeführt. In dieser in Operatoren implementierten Phase werden die zuvor in *ResultSets* sortierten Nexus Objekte parallel an die Server übergeben. Besitzt ein Nexus Objekt mehr als eine *NOL*, dann wird das Nexus Objekt an alle Server dieser *NOLs* übergeben. Die *CRL* Dokumente, welche von den Servern zurückgegeben werden, werden zu einem *CRL* Dokument zusammengefaßt. Wurde ein *insert* für ein Objekt auf mehr als einem Server ausgeführt und war der *insert* nicht bei allen Servern erfolgreich, dann ist beim Vereinigen der Antworten dieser Server ein *failed* die Antwort für dieses Objekt.

Nach der AWML-Phase 1 werden die erfolgreich eingefügten Nexus Objekte in ein *ResultSet* sortiert, welches in AWML-Phase 2 dem Cache übergeben wird. Für die nicht erfolgreich eingefügten Objekte wird für jeden Server an den dieses Nexus Objekt in AWML-Phase 1 übergeben wurde ein Eintrag mit der *NOL* dieses Nexus Objekts einem jeweiligen *restriction*-Ausdruck hinzugefügt. Die Objekte im *restriction*-Ausdruck werden wieder vom Server entfernt werden. Dabei kann allerdings nicht garantiert werden, daß alle Objekte erfolgreicht entfernt werden.

In der AWML-Phase 2 werden die Nexus Objekte dem Cache übergeben, falls notwendig, und Nexus Objekte auf den Servern entfernt, falls notwendig.

In einem *CRL* Dokument wird für alle Nexus Objekte die der Methode übergeben wurden zurückgegeben, ob die Methode für ein Nexus Objekt erfolgreich ausgeführt werden konnte.

4.4 Ausführen von update und delete



Den Methoden *update* und *delete* wird ein *AWQL* Dokument übergeben. Wie bei der Methode *query* wird zuerst der *restriction* Ausdruck in der ersten Phase in eine *DNF* konvertiert und in der zweiten Phase ein *RequestMatrix* Objekte aufgebaut. Der *CacheGetRequestMatrix* Operator wird allerdings nicht verwendet, da die *update* bzw. *delete* Operation nur auf den Servern durchgeführt werden soll.

Jetzt wird in der AWQL-Phase 4 die Anfragen an die *update* bzw. *delete* Methode der Server parallel gestellt und die *CRL* Antworten zusammengefaßt. Wurde eine Operation für ein Objekt auf mehr als einem Server ausgeführt und war die Operation nicht bei allen Servern erfolgreich, dann ist beim Vereinigen der Antworten dieser Server ein *failed* die Antwort für dieses Objekt.

Um ein *insert*, *update* oder *delete* sicher auf entweder allen oder keinen Servern ausführen zu können, sollten die *ACID* Eigenschaften einer solchen Transaktion erfüllt sein. Dazu sollte in einer zukünftigen Version ein 2-Phasen-Commit Protokoll für diese Operationen implementiert werden.

Für die aktualisierten bzw. gelöschten Nexus Objekte wird ein neuer *restriction* Ausdruck erstellt. Dieser *restriction* Ausdruck ist eine *oder* Verknüpfung von *equal* Vergleichen mit den *NOLs* der Objekte, welche erfolgreich aktualisiert oder gelöscht wurden. Mit diesem *restriction* Ausdruck wird eine Anfrage erstellt und diese dem Cache übergeben.

4.5 Die Phasen der Anfragen

Die drei Phasen der Verarbeitung eines *AWQL* Dokuments, die Phase in der Nexus Objekte bei der Verarbeitung eines *AWML* Dokuments an die Nexus Server übergeben werden und die Phase in der Nexus Objekte auf den Nexus Servern aktualisiert oder dort entfernt werden sollen jeweils mit Operatoren implementiert werden. Diese Operatoren werden von den ausführenden Methoden der Föderation in der jeweiligen Phase zu einem Baum zusammengesetzt.

4.5.1 AWQL-Phase 1

In dieser Phase wird der Baum unter einem *restriction* Element in eine DNF konvertiert und die DNF weiter zusammengefaßt.

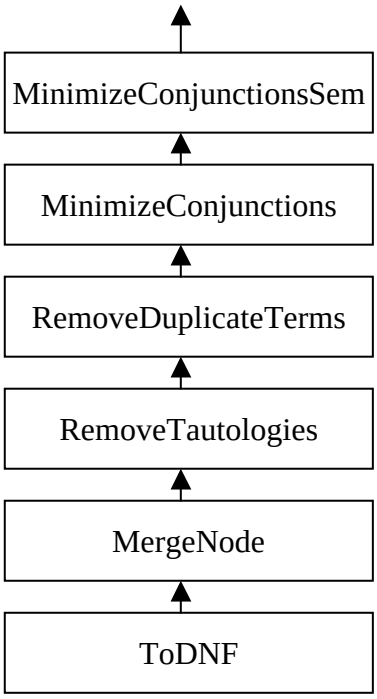


Abb. 8: DNF-Operatoren Baum

Die Operatoren dieser Phase besitzen die Methoden *get*. Die Methode *get* ruft die Methode *get* des Kindoperators auf. Dieser Kindoperator gibt den bisher bearbeiteten Baum in einem *RestrictionOperator* Objekt zurück. Der Operator bearbeitet den Baum und gibt dann ebenfalls das *RestrictionOperator* Objekt zurück.

Den Konstruktoren der Operatoren wird unter anderen ein Kindoperator und das *RestrictionOperator* Objekt des Wurzelknotens des noch nicht bearbeiteten *restriction* Ausdrucks übergeben.

Der unterste Operator im Baum ist der *ToDNF* Operator. Danach folgt die Operatoren *MergeNode*, *RemoveTautologies*, *RemoveDuplicateTerms*, *MinimizeConjunctions* und *MinimizeConjunctionsSem*. Diese Operatoren werden im Entwurf und im Feinentwurf beschrieben.

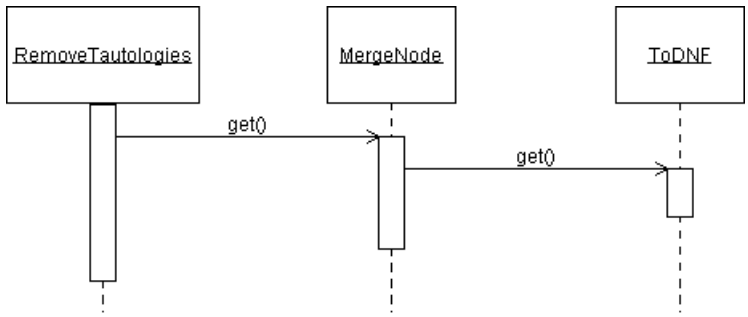


Abb. 9: Sequenzdiagramm eines Aufrufs von drei der DNF-Operatoren

4.5.2 AWQL-Phase 2

In dieser Phase werden die Nexus Server bestimmt, an die Anfragen zu stellen sind. Der *GetRequestMatrix* Operator ruft für jede Konjunktion der *DNF* einen *DetermineServers* Operator auf. Dieser Operator bestimmt die Server, an welche eine Anfrage mit der jeweiligen Konjunktion zu stellen ist. Der *GetRequestMatrix* Operator baut aus allen Antworten der *DetermineServers* Operatoren eine *RequestMatrix* auf und gibt diese an den *CacheGetRequestMatrix* Operator zurück. Der *CacheGetRequestMatrix* Operator übergibt diese *RequestMatrix* dem Cache. Der Cache gibt eine *RequestMatrix* als Antwort zurück. Diese *RequestMatrix* wird vom *CacheGetRequestMatrix* Operator zurückgegeben.

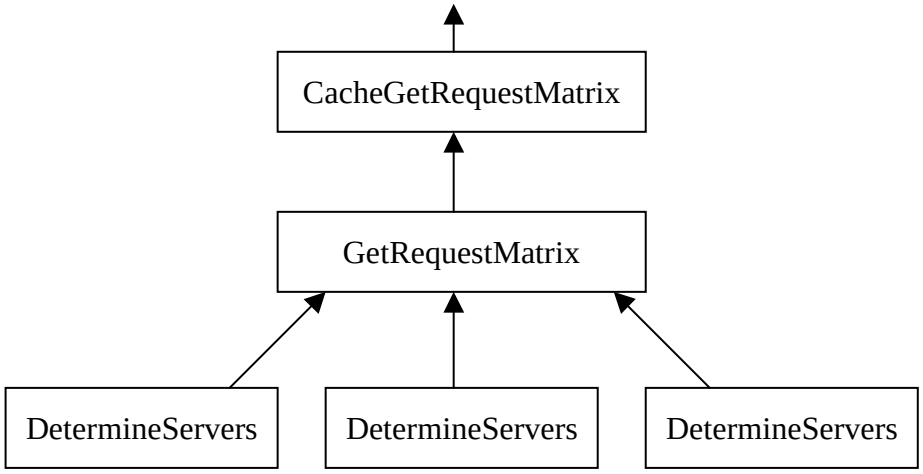


Abb. 10: Operatoren zum Aufbau der RequestMatrix

Die *DetermineServers* Operatoren führen ihren Algorithmus in einem Thread aus. Der *GetRequestMatrix* Operator ruft die *get* Methoden der *DetermineServers* Operatoren auf, um darauf zu warten, daß alle *DetermineServers* Operatoren ihren Algorithmus abgearbeitet haben.

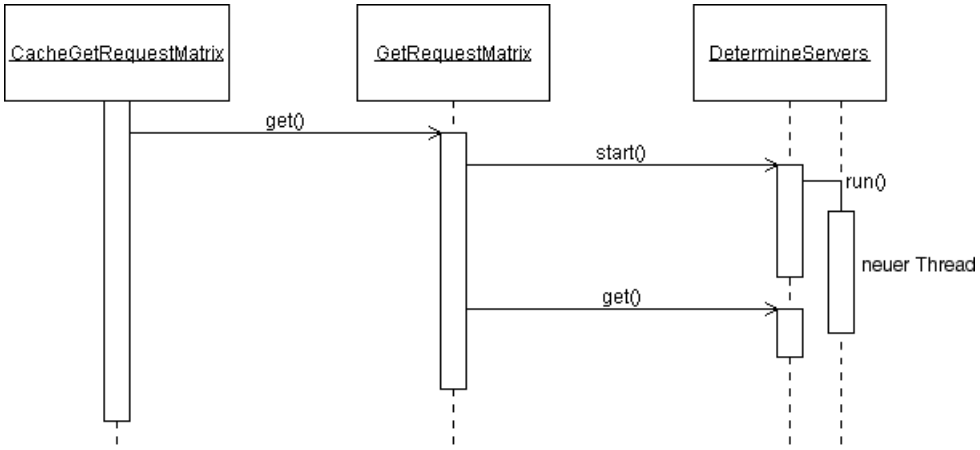


Abb. 11: Sequenzdiagramm der RequestMatrix Operatoren

4.5.3 AWQL-Phase 3

In dieser Phase werden die Anfragen an die Server gestellt und die Antworten in einem n-Wege-Merge zusammengefaßt.

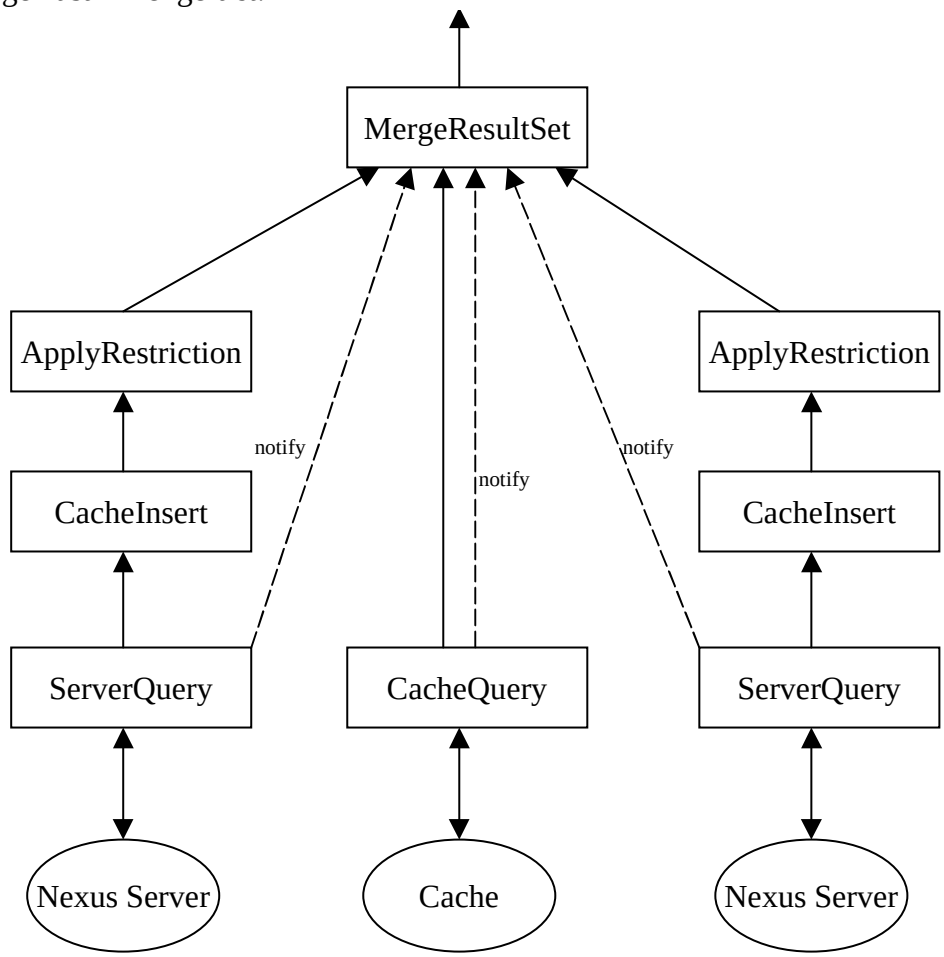


Abb. 12: Operatoren zum Stellen der Anfrage an die Server

Die Operatoren dieser Phase besitzen eine *start* und eine *get* Methode. Die *ServerQuery* und die *CacheQuery* Operatoren besitzen zusätzlich eine *setNotifyOperator* Methode. Der *MergeResultSet* Operator besitzt zusätzlich eine *notifyOperator* Methode.

Die *start* Methode ruft alle *start* Methoden der Kindoperatoren auf, soweit Kindoperatoren existieren, und kehrt dann zurück. Bei den *ServerQuery* und den *CacheQuery* Operatoren weckt die *start* Methode einen Thread dieser Operatoren auf. Diese Thread führt die Anfrage bei dem Server bzw. dem Cache durch.

Die Methode *get* eines Operators ruft die *get* Methode des Kindoperators auf um dessen *ResultSet* zu erhalten. Danach bearbeitet der Operator das *ResultSet*, wenn nicht *null* zurückgegeben wurde und gibt das bearbeitete *ResultSet* zurück. Die *ServerQuery* und *CacheQuery* Operatoren geben eine *null* Referenz zurück, wenn das Ergebnis der Anfrage noch nicht vorliegt. Tritt ein Fehler auf, dann gibt die Methode eine *FederationException* zurück.

Die get Methode des MergeResultSet Operators wartet solange bis die *ServerQuery* bzw. *CacheQuery* Operatoren alle eine Antwort (also ein *ResultSet* oder eine *SOAPException*) an den *MergeResultSet* Operator zurückgegeben haben und der *MergeResultSet* Operator alle *ResultSets* vereinigt hat. Danach wird das ResultSet zurückgegeben.

Hat ein *ServerQuery* oder *CacheQuery* Operator das Ergebnis der Anfrage erhalten und in ein *ResultSet* konvertiert, dann ruft er die *synchronized* Methode *notifyOperator* des *MergeResultSet* Operators auf. Diese Methode weckt den Thread auf und veranlaßt ihn das *ResultSet* vom entsprechenden Kindoperator zu holen.

Die *ServerInsert* Operatoren übergeben das *ResultSet* und den *Request* an den Cache.

Die *ApplyRestriction* Operatoren wenden den *restriction* Ausdruck auf das *ResultSet* an, sollte dies notwendig sein.

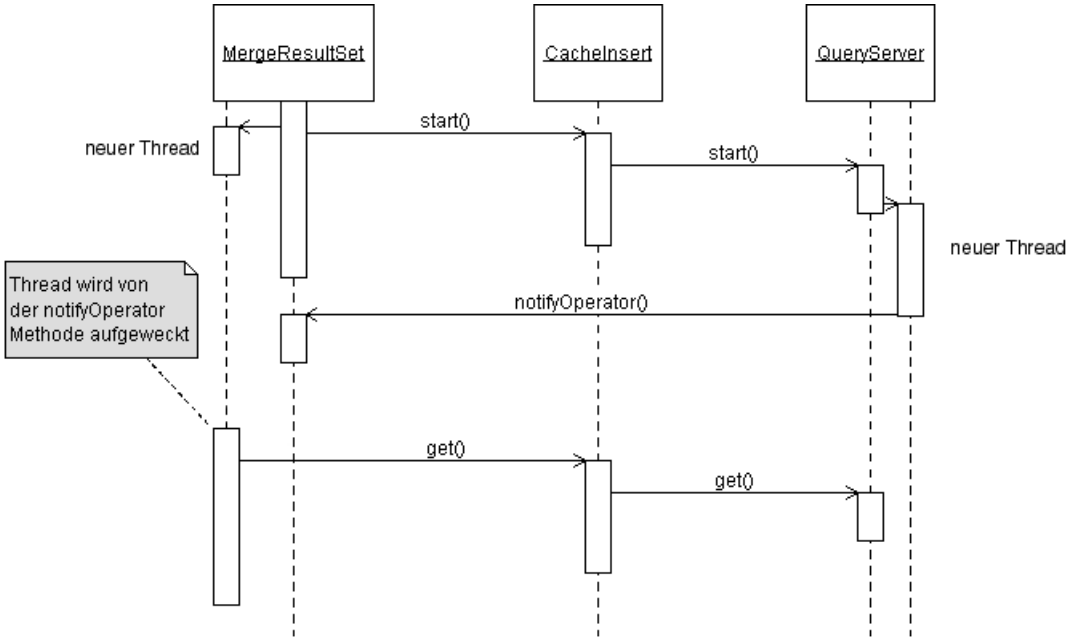


Abb. 13: Sequenzdiagramm der Query Operatoren

4.5.4 AWML-Phase 1: Übergeben der AWML Dokumente

Diese Phase wird von der *insert* Methode der Föderation ausgeführt. In dieser Phase werden die Nexus Objekte an die *insert* Methoden der Nexus Server übergeben. Die Operatoren dieser Phase besitzen eine *start* und eine *get* Methode. Der *ServerInsert* Operator besitzt zusätzlich eine *setNotifyOperator* Methode. Der *MergeCRL* Operator besitzt zusätzlich eine *notifyOperator* Methode.

Die Methode *get* ruft alle *get* Methoden der Kindoperatoren bzw. die *get* Methode des Kindoperators auf. Diese Kind-Operatoren geben jeweils ein *CRLSet* Objekt zurück. Der Operator bearbeitet das *CRLSet* Objekt und gibt ein neues *CRLSet* Objekt zurück.

Der *MergeCRL* Operator wartet darauf, bis ihn einer der *ServerInsert* Operatoren aufweckt, indem der Kindoperator die *notifyOperator* Methode des *MergeCRL* Operators aufruft. Danach wird das *CRLSet* Objekt vom Kindoperator geholt und vereinigt.

Die untersten Operatoren des Baumes sind die *ServerInsert* Operatoren. Der nächste Operator ist der *MergeCRL* Operator.

4.5.5 AWML-Phase 2: Übergeben an Cache und entfernen von Objekten, falls nötig.

In dieser Phase werden die erfolgreich eingefügten Nexus Objekte dem Cache übergeben und die auf möglicherweise nicht allen Servern erfolgreich eingefügten Objekte wieder bei den Nexus Servern entfernt. Dabei kann allerdings nicht garantiert werden, daß die Objekte in diesem Fall bei allen Servern erfolgreich entfernt werden können, falls zum Beispiel die Verbindung zu einem Server nach dem *insert* und vor einem *delete* unterbrochen wird.

Um Objekte im Cache einzufügen wird der *ServerInsert* Operator verwendet. Die *get* Methode dieses Operators gibt *null* zurück, da eine mögliche Antwort des Cache nicht benötigt wird. Entfernt werden die Objekte mit den *ServerDelete* und *MergeCRL* Operatoren.

4.5.6 AWQL-Phase 4: Update bzw. Delete auf Servern ausführen

In dieser Phase werden die *AWQL* Anfragen an die Methoden *update* oder *delete* der Server übergeben. Dazu werden die Operatoren *ServerUpdate* bzw. *ServerDelete* und *MergeCRL* verwendet. Alle erfolgreich aktualisierten oder gelöschten Objekte werden in einer neuen *AWQL* Anfrage mit den *ServerUpdate* bzw. Server*Delete* Operatoren dem Cache übergeben.

4.6 Bestimmen der anzufragenden Server bei einer query Anfrage

Nachdem der *restriction*-Ausdruck in eine DNF konvertiert wurde, muß bestimmt werden an welche Server Anfragen zu stellen sind. Dazu werden für jede Konjunktion die Server bestimmt, welche für diese Konjunktion Nexus Objekte liefern können.

Dazu wird jede Konjunktion überprüft und folgender Algorithmus ausgeführt:

- Zuerst werden alle nicht-invertierten “*type*” *equal*-Vergleiche gesucht. Die Typen jedes dieser *in* oder *equal* Knoten der Konjunktion werden mit den anderen Typen der Konjunktion logisch *and*-verknüpft. Dabei wird für jeden bereits gefundenen Typ überprüft, ob dieser ein Sub-Typ der neu gefundenen Typen ist. Ist er das nicht und sind auch keine der neu gefundenen Typen Sub-Typen von diesem Typ, dann wird er gelöscht. Sind ein oder meh-

rere neu gefundene Typen ein Sub-Typ von diesem Typ, dann wird dieser Typ gelöscht und durch die Sub-Typen ersetzt.

- Die Typen der invertierten "*type*" equal-Vergleiche werden in einer Liste eingetragen.
- Die nicht invertierten "*nol*" Vergleiche werden gesucht und die *NOLs* miteinander *and*-verknüpft. Für jede bereits gefundene *NOL* wird überprüft, ob sie in der Liste mit neuen *NOLs* enthalten ist. Ist das nicht der Fall, dann wird sie entfernt.
- Die *NOLs* der invertierten "*nol*" Vergleiche werden in einer Liste eingetragen.
- Die gefundenen nicht invertierten Polygone werden per *union* miteinander vereinigt.
- Nachdem alle Elemente der Konjunktion geparst wurden, werden für alle gefundenen nicht invertierten *NOLs* überprüft, ob sie in der Liste mit invertierten *NOLs* vorkommen. Ist das der Fall, dann werden sie entfernt.
- Wurden *NOLs* gefunden, dann werden die Server über die *SpaSeLocators* der *NOLs* bestimmt.
- Wurden keine *NOLs* gefunden, dann werden die Server über gefundene Polygone bzw. gefundene Typen bestimmt. Dazu wird eine Anfrage mit dem Multipolygon und den Typen an das *ASR* gestellt.
- Wurden in der Konjunktion invertierte Typen gefunden, dann hat das *ASR* eventuell mehrere Server zurückgegeben, die für diese Konjunktion keine Nexus Objekte liefern können. Diese werden in diesem Fall jetzt identifiziert und entfernt. Dazu wird für jeden Server und für jeden seiner Typen überprüft, ob dieser jeweilige Type ein Kind-Typ eines invertierten Typs ist. Ist das der Fall, dann wird dieser Typ aus der Liste seines Servers, welche vom ASR zurückgegeben wurde gelöscht. Wurden für einen gefundenen Server alle Vergleiche durchgeführt und sind keine Typen in seiner Liste mehr vorhanden, dann kann dieser Server der Konjunktion keine Nexus Objekte liefern. Er wird entfernt.

4.7 Aufteilen von Klassen in Aufgabengebiete

Es werden also für folgende Aufgaben Klassen benötigt:

- Die Föderation
- Operatoren zum Konvertieren des Restriction Ausdrucks in eine *DNF*
- Operatoren zum bestimmen der Server, an welche Requests zu senden sind und Operatoren zur Kommunikation mit dem Cache.
- Operatoren zum Übergeben eines *query* Requests an Server und zum Zusammenfassen von Antworten
- Operatoren zum Übergeben von *update*, *insert* und *delete* Requests an Server und zum Zusammenfassen von den zurückgegebenen CRL Dokumenten.
- Geoklassen, um Polygone beim Konvertieren des Restriction Ausdrucks in eine DNF zusammenfassen zu können.
- Sonstige Klassen (Objekt, welches an den Cache übergeben und vom Cache zurückgegeben wird, Cache für Schemata, der Cache, usw.)

5 Entwurf

Im Entwurf sollen die noch zu implementierenden Algorithmen (bzw. Methoden) betrachtet werden. Auf die bereits vorhandenen Klassen wird nicht näher eingegangen.

Es werden folgende Methoden entworfen:

- Die query, update, insert und delete Methoden der Föderation.
- Die *getCRL* Methode der Föderation, welche von der *update* und der *delete* Methode der Föderation verwendet wird.
- Die Methoden der *DNF* Operatoren, welche einen *restriction* Ausdruck in eine *DNF* konvertieren.
- Die Methoden des Operators welcher die Server bestimmt, an die Anfragen zu stellen sind, der Operator welcher die *RequestMatrix* aus den gefundenen Anfragen zusammensetzt und der Operator, welcher die erstellten Anfragen dem *Cache* übergibt, damit dieser eine Menge von neuen Anfragen in einem *RequestMatrix* Objekt zurückgibt.
- Die Operatoren, welche Anfragen an *Spatial Model Server* oder den *Cache* stellen und *ResultSets* oder *CRLSets* zurückgeben.
- Die Operatoren, welche die Antworten von Servern, also *ResultSets* oder *CRLSets* zusammenfassen.
- Die Operatoren, welche ein *ResultSet* sortieren, es dem *Cache* übergeben, oder einen *restriction* Ausdruck auf ein *ResultSet* anwenden.

Die Hilfsklassen, welche zum Beispiel die Kommunikation mit den Nexus Servern per *ApacheSOAP* implementieren oder ein *AWQLQuery* Objekt in ein *AWQL* Dokument konvertieren werden hier nicht näher beschrieben (Siehe “Feinentwurf”).

5.1 Die Methoden der Föderation

5.1.1 query

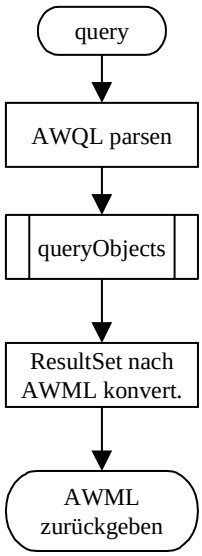


Abb. 14: Flußdiagramm von query

Um eine Anfrage an die *query* Methode der Föderation zu beantworten, muß das *AWQL* Dokument geparst werden, um die Objekte, welche die Bedingungen des *AWQL* Dokuments erfüllen zu bestimmen. Dazu wird, nachdem das *AWQL* Dokument in ein *AWQLQuery* Objekt konvertiert wurde, die Methode *query* aufgerufen und das *AWQLQuery* Objekt übergeben. Diese Methode bearbeitet die Anfrage und gibt ein *ResultSet* mit den Nexus Objekten zurück. Das *ResultSet* wird in ein *AWML* Dokument konvertiert und zurückgegeben.

5.1.2 query

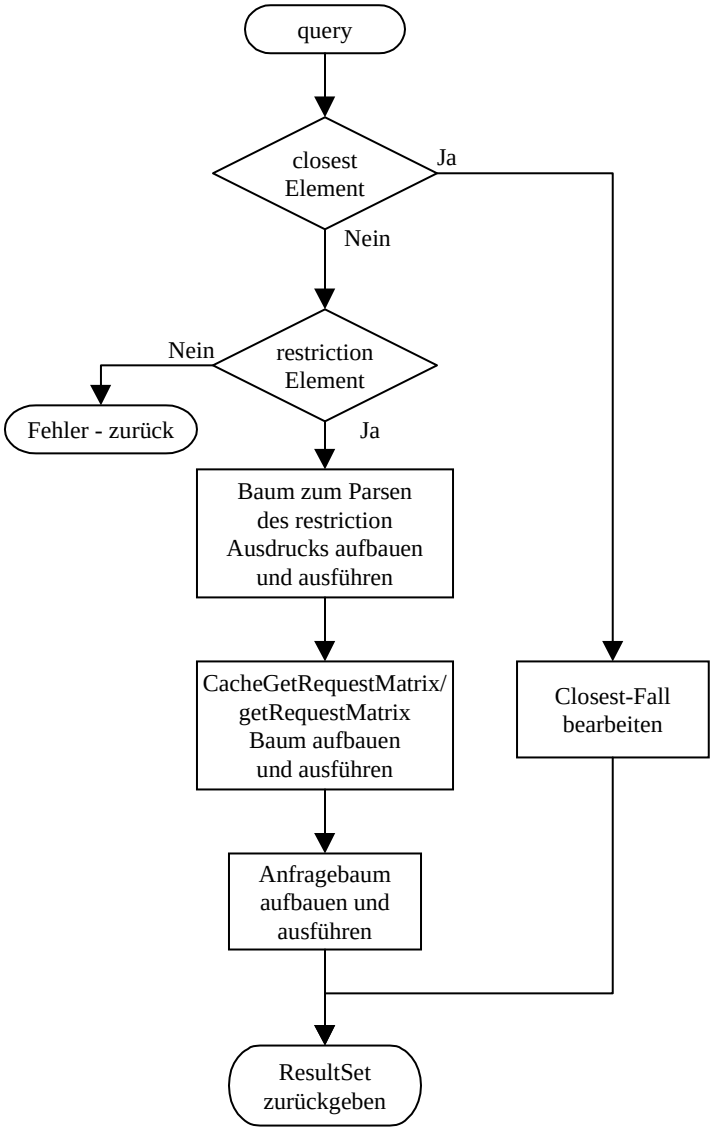


Abb. 15: Flußdiagramm von query

Ein *AWQL* Dokument kann unter anderen ein *restriction* Element und ein *closest* Element enthalten. Diese dienen mit dazu, die Quellen der Objekte und welche Bedingungen diese erfüllen müssen zu bestimmen.

Zuerst wird überprüft, ob ein *closest* Element enthalten ist. Ist das der Fall, dann muß das *closest* Element beim Verarbeiten der Anfrage berücksichtigt werden. Dieser Fall wird im Rahmen

dieser Diplomarbeit nicht berücksichtigt.

Danach wird überprüft, ob ein *restriction* Element enthalten ist. Wenn dies der Fall ist, dann wird der boolsche Ausdruck des *restriction* Elements zur Bestimmung der Quellen der Objekte verwendet.

Dazu wird die AWQL-Phase 1 ausgeführt. Zuerst wird der boolsche Ausdruck in eine *DNF* konvertiert und Konjunktionen, falls möglich, zusammengefaßt. Dazu werden die Operatoren verwendet, welche den *restriction* Ausdruck in eine *DNF* konvertieren und diesen zusammenfassen. Aus diesen Operatoren wird ein Anfrage-Umschreib-Baum aufgebaut, wobei der Operator *ToDNF* der unterste Operator in diesem Anfragebaum ist. Die *get* Methode des Anfrage-Umschreib-Baums wird ausgeführt, um die *DNF* zu erhalten.

Nachdem die *DNF* aufgebaut ist, muß in der AWQL-Phase 2 bestimmt werden, an welche Server welches *AWQL* Dokument übergeben werden muß. Dazu wird ein Server-Such-Baum aufgebaut. Der *GetRequestMatrix* Operator bestimmt für jede Konjunktion, an welche Server die jeweilige Konjunktion, oder die DNF übergeben werden soll. Dazu verwendet der *GetRequestMatrix* Operator für jede Konjunktion einen *DetermineServers* Operator.

Um den Cache einzubinden, wird dem Server-Such-Baum ein *CacheGetRequestMatrix* Operator hinzugefügt.

Danach wird die *get* Methode des obersten Operators des Server-Such-Baums aufgerufen, um gegebenenfalls die Anfragen an das *ASR* zu starten und das *RequestMatrix* Objekt zu erhalten.

Um die Anfragen an die Server auszuführen, wird in der AWQL-Phase 3 ein Anfrage-Ausführ-Baum aufgebaut. Für jeden Server des *RequestMatrix* Objekts wird ein *ServerQuery* Operator instanziiert und falls notwendig ein *CacheInsert* und *ApplyRestriction* Operator. Diese Operatoren werden einem *MergeResultSet* Operator übergeben und die Anfrage wird ausgeführt.

Das von der *get* Methode des *MergeResultSet* Operators im Anfrage-Ausführ-Baum erhaltene *ResultSet* wird zurückgegeben.

5.1.3 update

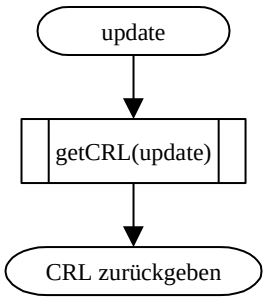


Abb. 16: Flußdiagramm von update

Um ein Update auszuführen, wird die Methode *getCRL* aufgerufen. Der Methode wird der Parameter "*update*" übergeben, damit diese Methode die Updates an die richtige Methode wei-

tergibt.

5.1.4 insert

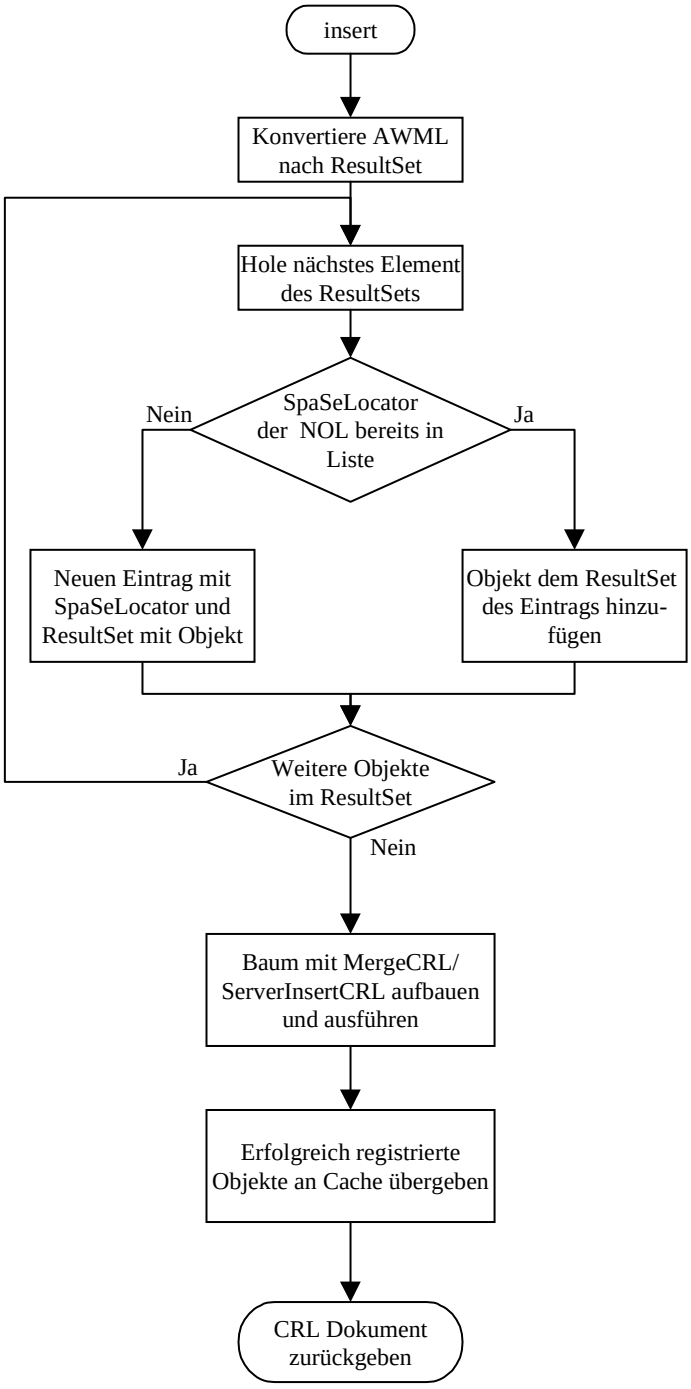


Abb. 17: Flußdiagramm von insert

Wird der Methode *insert* der Föderation ein *AWML* Dokument übergeben, dann muß die Föderation die Objekte des Dokuments an die *insert* Methoden der richtigen Server weiterleiten. Diese werden durch die *SpaSeLocators* der *NOLs* bestimmt. Die *CRL* Dokumente, die von den

Servern zurückgegeben werden, müssen vereinigt werden und von der Methode *insert* zurückgegeben werden.

Dazu wird zuerst das *AWML* Dokument in ein *ResultSet* konvertiert. Für jedes Objekt des *ResultSets* werden die *SpaSeLocator* der *NOLs* überprüft.

Ist unter diesem *SpaSeLocator* bereits ein *ResultSet* in einer HashMap vorhanden, dann wird das aktuelle Objekt dem *ResultSet* dieses *SpaSeLocators* hinzugefügt.

Ist noch kein Eintrag vorhanden, dann wird ein neues *ResultSet* erzeugt und das aktuelle Objekt diesem *ResultSet* hinzugefügt. Das *ResultSet* wird unter dem *SpaSeLocator* in der HashMap eingetragen.

Nachdem alle Objekte geparst wurden, wird in der AWML-Phase 1 für jeden Eintrag in der *HashMap* ein *ServerInsert* Operator instanziiert und diesem das *ResultSet* und der *SpaSeLocator* übergeben. Alle *ServerInsert* Operatoren werden einem *MergeCRL* Operator übergeben.

Der Anfragebaum wird gestartet und dann die Methode *get* aufgerufen, um die *CRL* zu erhalten.

Diese *CRL* wird zurückgegeben werden.

Nachdem der *MergeCRL* Operator die *CRL* zurückgegeben hat, werden alle Objekte, für welche die Antwort in der *CRL false* ist aus dem *ResultSet* entfernt. Dieses *ResultSet* wird dann einem *ServerInsert* Operator übergeben. Die NOLs der Objekte, für welche die Antwort *false* ist, werden in oder-verknüpfte *RestrictionOperator* Bäume eingetragen, wobei die *NOLs* in jedem Baum jeweils denselben *SpaSeLocator* besitzen. Für diese Bäume werden *Request* Objekte aufgebaut.

In der AWML-Phase 2 werden diese *Request* Objekte *ServerDelete* Operatoren übergeben und das *ResultSet* und der *Cache* werden einem *ServerInsert* Operator übergeben. Die Server Operatoren werden einem MergeCRL Operator übergeben.

Der Anfragebaum wird gestartet und dann die Methode *get* aufgerufen, um die *CRL* zu erhalten.

5.1.5 delete

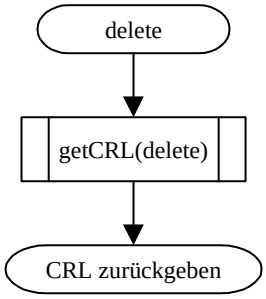


Abb. 18: Flußdiagramm von delete

Um ein Delete auszuführen, wird die Methode *getCRL* aufgerufen. Der Methode wird der Parameter "*delete*" übergeben, damit diese Methode die updates an die richtige Methode weitergibt.

5.1.6 getCRL

Wenn ein *update* oder *delete* Request der Föderation übergeben wird, dann muß die Föderation ein *AWQL* Dokument parsen. Ist in dem *AWQL* Dokument ein *closest* Element vorhanden, dann muß iterativ auf den Servern gesucht werden, wobei bei jeder Iteration das Gebiet in dem gesucht werden muß vergrößert wird, bis genügend Nexus Objekte gefunden wurden, welche die Bedingung des *AWQL* Dokument zu erfüllen. Diese müssen dann entweder entfernt oder aktualisiert werden. Wenn ein *closest* Element in der Anfrage vorhanden ist, dann müssen also zuerst die Nexus Objekte wie bei einem *query* Request gesucht werden. Danach muß ein *update* oder *delete* Request explizit für die gefundenen Objekte ausgeführt werden. Dies könnte zum Beispiel so implementiert werden:

getCRL für Closest Element in Anfrage:

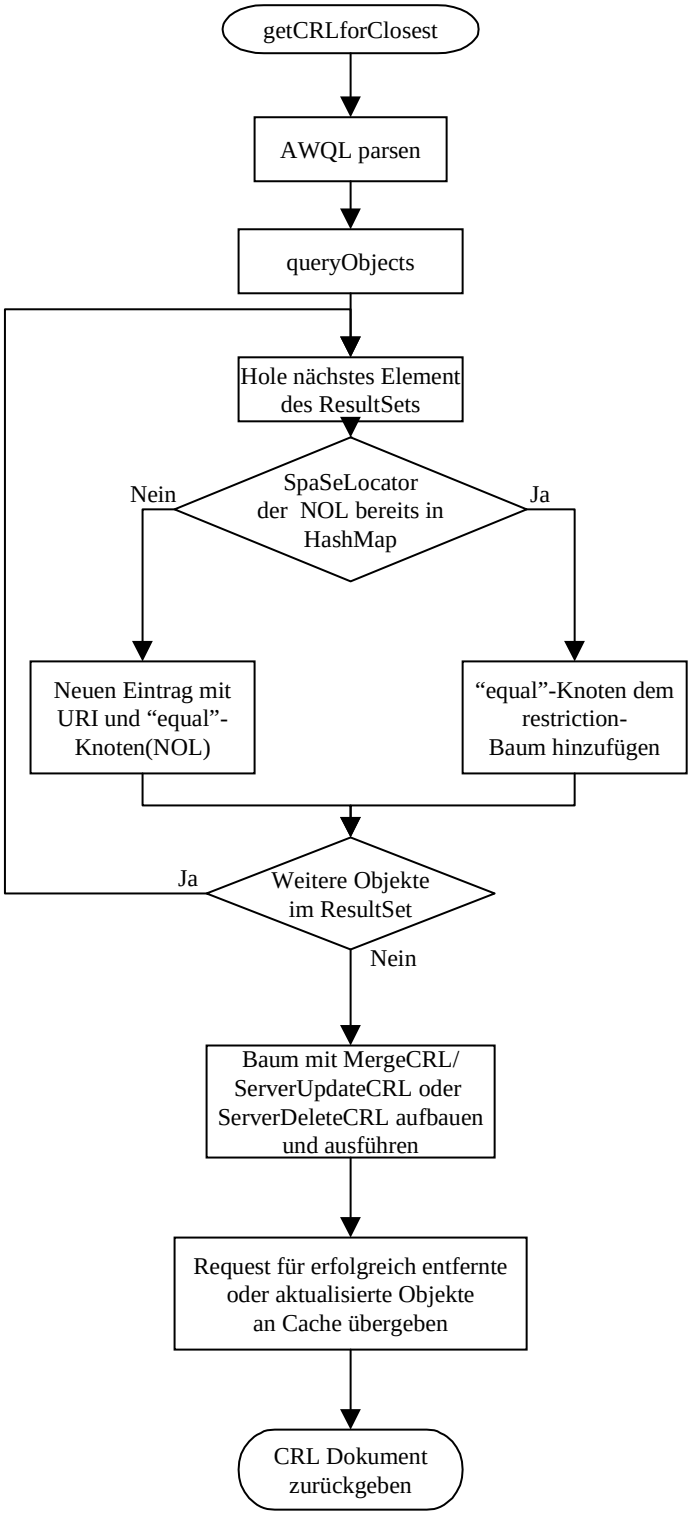


Abb. 19: Flußdiagramm von getCRL für Closest Element in Anfrage

Die Methoden *update* und *delete* rufen diese Methode auf. Nachdem das *AWQL* Dokument geparst wurde, ruft diese Methode zuerst für das *AWQL* Dokument die Methode *queryObjects*

auf, um die Objekte zu holen, für welche die Operation *update* oder *delete* ausgeführt werden soll.

Diese Objekte werden von der Methode *queryObjects* in einem *ResultSet* zurückgegeben.

Für jedes Objekt des *ResultSets* werden die *SpaSeLocators* der *NOLs* überprüft.

Ist unter diesem *SpaSeLocator* bereits ein Eintrag in einer *HashMap* vorhanden, dann wird für das aktuelle Objekt des *ResultSets* dem enthaltenen *RestrictionOperator* ein *equal* Knoten mit dieser *NOL* hinzugefügt.

Ist noch kein Eintrag vorhanden, dann wird ein neuer *equal RestrictionOperator* mit der *NOL* des Nexus Objekts erzeugt und unter dem *SpaSeLocator* in der *HashMap* eingetragen.

Danach wird aus dem geparsten *ResultSet* ein *AWQL* Dokument aufgebaut, wobei der *restriction* Abschnitt eine Disjunktion von *equal* Knoten mit den *NOLs* oder ein *equal* Knoten mit einer *NOL* ist. Dieses *AWQL* Dokument wird zusammen mit dem *SpaSeLocator* des Cache einem *ServerUpdate* Operator übergeben. Dieser Operator wird gestartet, ohne auf die Antwort des Cache zu warten.

Danach wird für jeden *RestrictionOperator* aus der *HashMap* ein *AWQL* Dokument erzeugt und einem *ServerUpdate* oder *ServerDelete* Operator übergeben. Dem Operator wird auch der *SpaSeLocator* des Eintrags übergeben. Alle Operatoren werden einem *MergeCRL* Operator übergeben.

Der Anfragebaum wird gestartet und das erhaltene *CRL* Dokument zurückgegeben.

Da die in dieser Diplomarbeit zu implementierende Föderation das *closest* Element (noch) nicht bearbeitet, wird folgender Algorithmus verwendet.

Algorithmus von getCRL:

Zuerst wird wie in *query* in AWQL-Phase 1 eine *DNF* aufgebaut und nach den Servern, an welche Anfragen zu stellen sind in AWQL-Phase 2 gesucht. Dabei wird der *CacheGetRequestMatrix* Operator allerdings nicht verwendet. Danach wird in AWQL-Phase4 anstatt des Anfrage-Ausführ-Baums ein Hole-CRL-Baum aufgebaut. Dieser übergibt die jeweiligen Anfragen an die Server, sammelt die *CRL* Antworten ein und fügt sie zu einem *CRL* Objekt zusammen. Dazu werden die Operatoren *ServerUpdate* bzw. *ServerDelete* und *MergeCRL* verwendet.

Der Baum wird aufgebaut und gestartet.

Die *get* Methode des *MergeCRL* Operator gibt das *CRL* Dokument zurück.

Aus den *NOLs* der Objekte, welche erfolgreich entfernt bzw. aktualisiert wurden, wird dann ein *RestrictionOperator* Baum aufgebaut. Mit diesem Baum wird eine neues *AWQLQuery* Objekt instanziiert und dieses Objekt und der *Cache* werden einem *ServerUpdate* bzw. *ServerDelete* Operator übergeben. Dieser Operator wird gestartet.

Das *CRL* Dokument wird zurückgegeben.

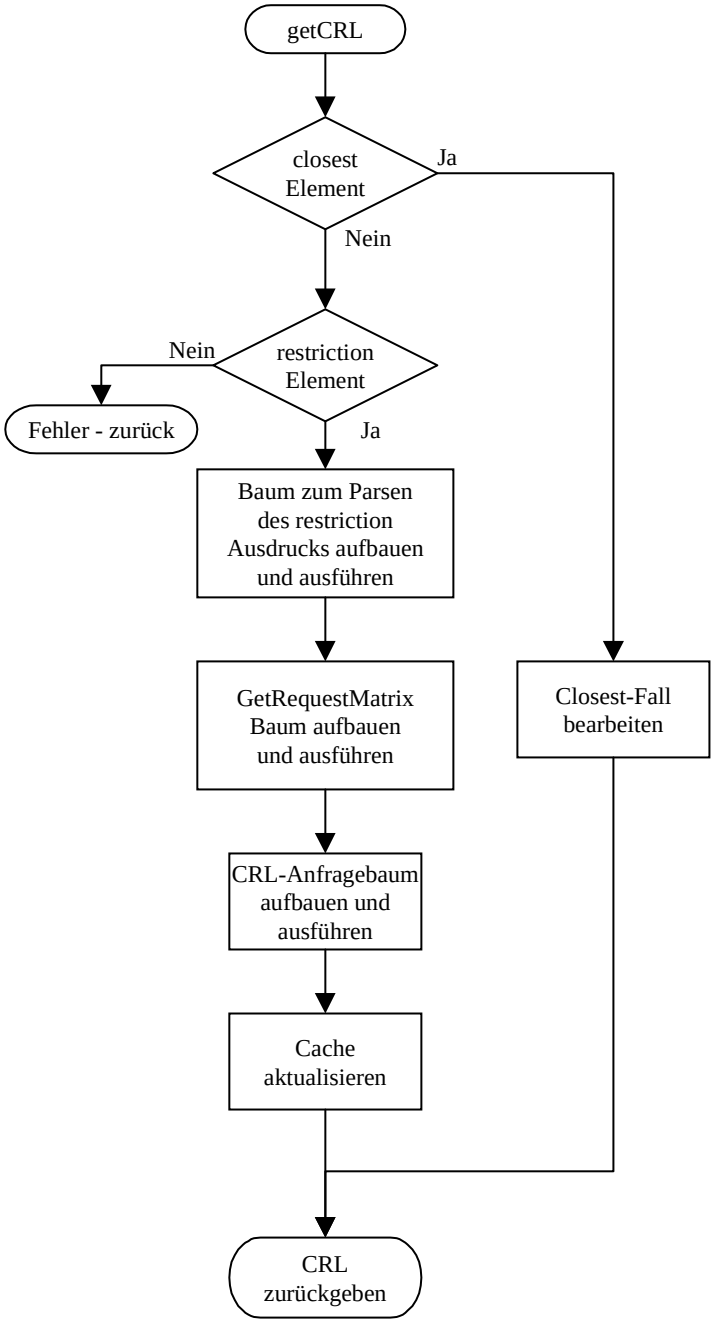


Abb. 20: Flußdiagramm von getCRL

5.2 Die Algorithmen zum Konvertieren eines restriction Ausdrucks in eine DNF:

5.2.1 ToDNF

Teil 1: Negation zu Literalen verschieben

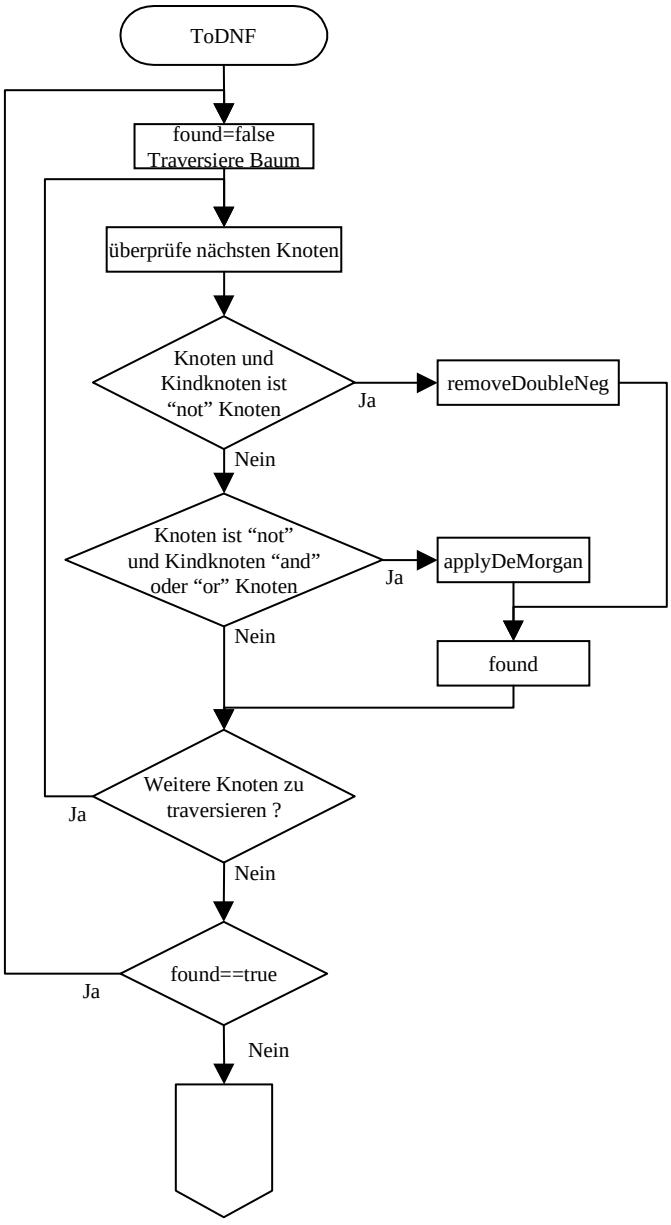


Abb. 21: Flußdiagramm von ToDNF

Teil 2: DNF durch ausmultiplizieren herstellen

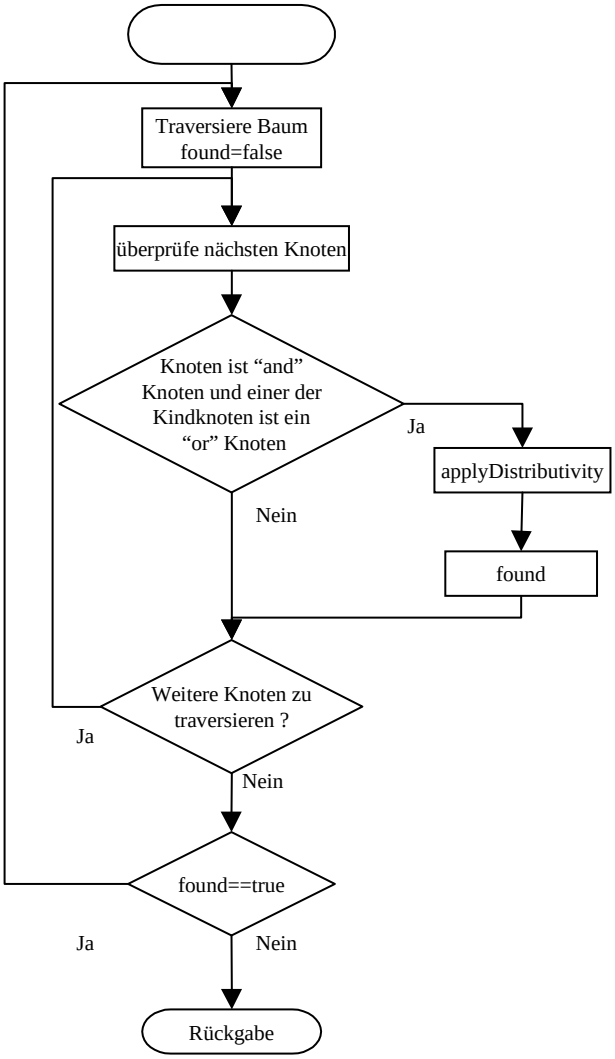


Abb. 22: Flußdiagramm von ToDNF

Dieser Operator konvertiert den boolschen Ausdruck des *restriction* Ausdrucks in eine DNF, also in eine Disjunktion von Konjunktionen.

Für die Methode des Operators, welche eine *DNF* zurückgibt wird folgender Algorithmus implementiert, wie er analog für die Konvertierung in eine *KNF* in OpenGIS Simple Features Specification for CORBA Revision 1.0 http://www.opengis.org/docs/99-054.pdf angegeben wird.

Dies erfolgt in zwei Phasen. In der ersten Phase werden die Doppelnegationen vor den Literalen entfernt und durch Anwenden der deMorganschen Regeln dafür gesorgt, daß nur noch Literale negiert sind:

(not (not G)) --> G
(not (and G H)) --> (or (not G) (not H))
(not (or G H)) --> (and (not G) (not H))

Dazu wird beginnend mit dem Wurzelknoten überprüft, ob der aktuelle und dessen Kindknoten ein *not* Knoten ist. Ist das der Fall, dann muß diese Doppelnegation entfernt werden, und es wird dazu *removeDoubleNeg* aufgerufen.

Danach wird überprüft, ob der aktuelle Knoten ein *not* Knoten ist und dessen Kindknoten ein *and* oder *or* Knoten ist. Ist das der Fall, dann muß die deMorgansche Regel angewandt werden, indem *applyDeMorgan* aufgerufen wird.

Immer dann, wenn entweder *removeDoubleNeg* oder *applyDeMorgan* aufgerufen wird, wird ein Flag *found* gesetzt.

Wurden alle Knoten geparst und ist das Flag gesetzt, dann wird das Flag gelöscht und der Baum erneut geparst, um sicher zu stellen, daß keine der beiden Regeln mehr auf den Baum anwendbar ist.

In der zweiten Phase wird durch Anwenden der folgenden Regeln (Distributivität) eine *DNF* erzeugt:

(and F (or G H)) --> (or (F and G) (F and H))
(and (or F G) H)) --> (or (F and H) (G and H))

Dazu wird der Baum wie in Phase 1 geparst. Für jeden Knoten wird geprüft, ob der Knoten ein *and* Knoten ist, und ob einer der Kindknoten ein *or* Knoten ist. Ist das der Fall, dann wird *applyDistributivity* aufgerufen.

Immer dann, wenn *applyDistributivity* aufgerufen wird, wird ein Flag *found* gesetzt.

Wurden alle Knoten geparst und ist das Flag gesetzt, dann wird das Flag gelöscht und der Baum erneut geparst, um sicher zu stellen, daß keine der beiden Regeln mehr auf den Baum anwendbar ist.

5.2.1.1 removeDoubleNeg

Diese Methode entfernt eine Doppelnegation aus dem boolschen Ausdruck, indem die Referenzen im Baum angepaßt werden.

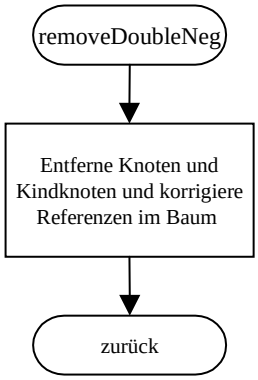


Abb. 23: Flußdiagramm von removeDoubleNeg

Beim Anpassen der Referenzen im Baum müssen die *parent* und *subOperators* (*Operands*) Referenzen so geändert werden, daß die beiden *not BooleanOperator Operatoren* nicht mehr im Baum sind. Sollte der Baum deshalb einen neuen Wurzel RestrictionOperator haben, dann müssen auch alle Referenzen auf den root-Operator aktualisiert werden. Die Referenzen auf den Wurzelknoten sind der Übersichtlichkeit wegen nicht dargestellt. Die gestrichelten Pfeile sind die alten Referenzen bzw. eine andere Referenz:

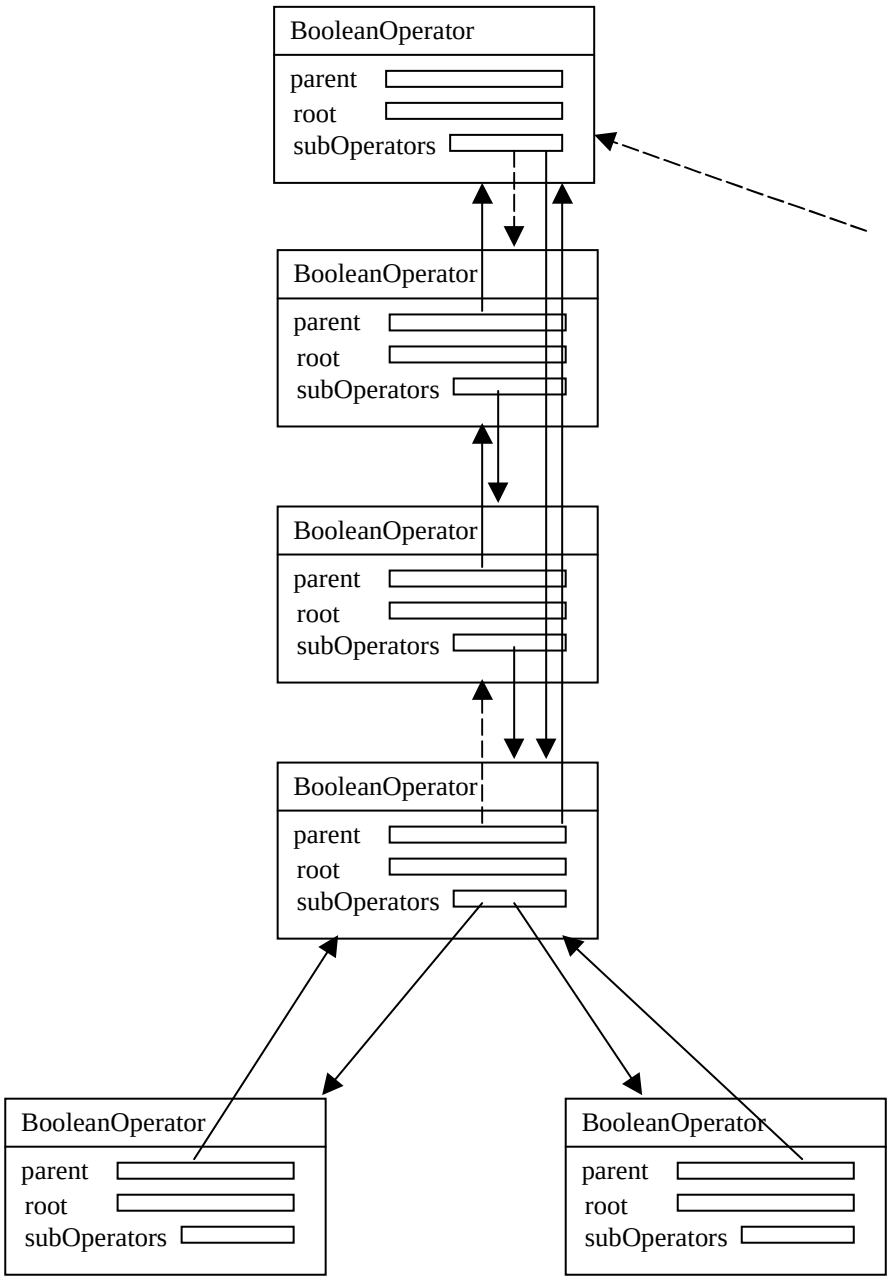


Abb. 24: Operator-Baum eines Restriction-Ausdrucks

5.2.1.2 applyDeMorgan

Diese Methode wendet die deMorgansche Regel an. Dazu wird zuerst der aktuelle *not* Knoten aus dem Baum entfernt. Ist der Kindknoten des entfernten Knotens ein *and* Knoten, dann wird ein *or* Knoten in den Baum eingefügt und ist der entfernte Knoten ein *or* Knoten, dann wird ein *and* Knoten eingefügt. Für jeden Kindknoten des *and* bzw. *or* Knotens wird ein *not* Knoten eingefügt und die Referenzen im Baum angepaßt.

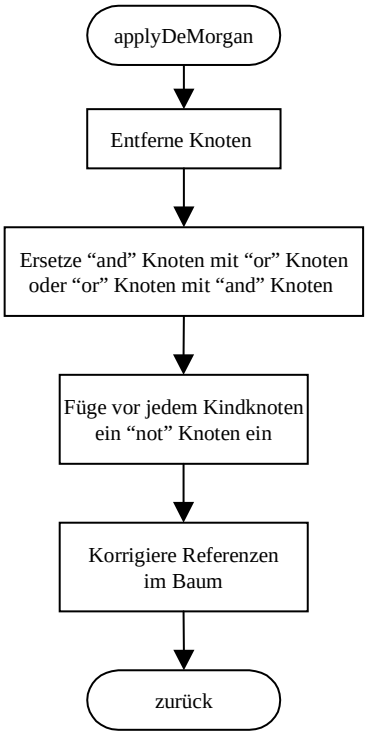


Abb. 25: Flußdiagramm von applyDeMorgan

5.2.1.3 applyDistributivity

Diese Methode wendet den Distributivitätssatz an, um eine *DNF* aufzubauen. Dazu wird der aktuelle *and* Knoten entfernt. Danach wird ein *or* Kindknoten gesucht. Für jeden Kindknoten dieses *or* Knotens wird ein *and* Knoten angelegt (der jeweilige Kindknoten ist Kindknoten dieses neuen *and* Knotens). Für die restlichen Kindknoten des zuvor entfernten *and* Knotens wird die Methode *duplicate* aufgerufen, um Kopien von diesem Teilbaum zu erstellen. Diese Teilbäume werden jedem zuvor angelegten *and* Knoten hinzugefügt. Zum Schluß wird ein *or* Knoten mit den zuvor angelegten *and* Knoten als Kindknoten angelegt und an den Baum angehängt. Danach werden noch die Referenzen im Baum korrigiert.

.

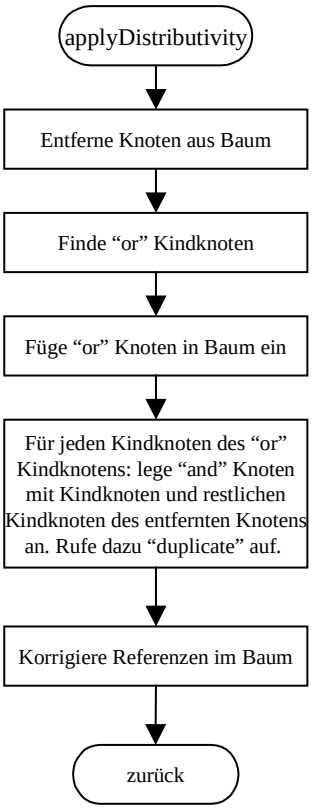


Abb. 26: Flußdiagramm von applyDistributivity

5.2.2 duplicate

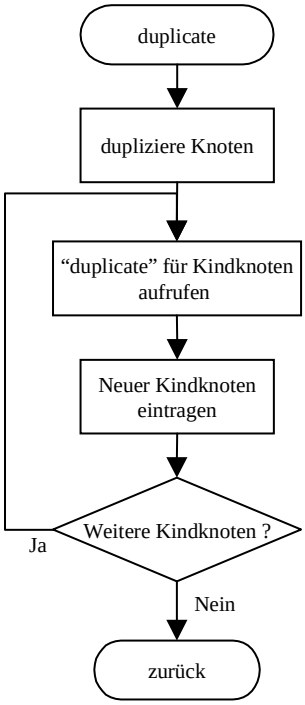


Abb. 27: Flußdiagramm von duplicate

Diese Methode kopiert einen Teilbaum des *restriction* Ausdrucks. Sie wird unter anderen von der Methode *applyDistributivity* verwendet.

Zuerst wird der übergebene Knoten dupliziert. Dazu wird, abhängig davon, ob es sich um einen *and*, *or*, *not*, *inside*, *overlaps*, *equal* oder *in* Knoten handelt ein solcher neuer Knoten angelegt und für alle eventuell vorhandenen Kindknoten die Methode *duplicate* rekursiv aufgerufen. Die Methode gibt den Wurzelknoten des neuen Teilbaums zurück.

5.2.3 MergeNode

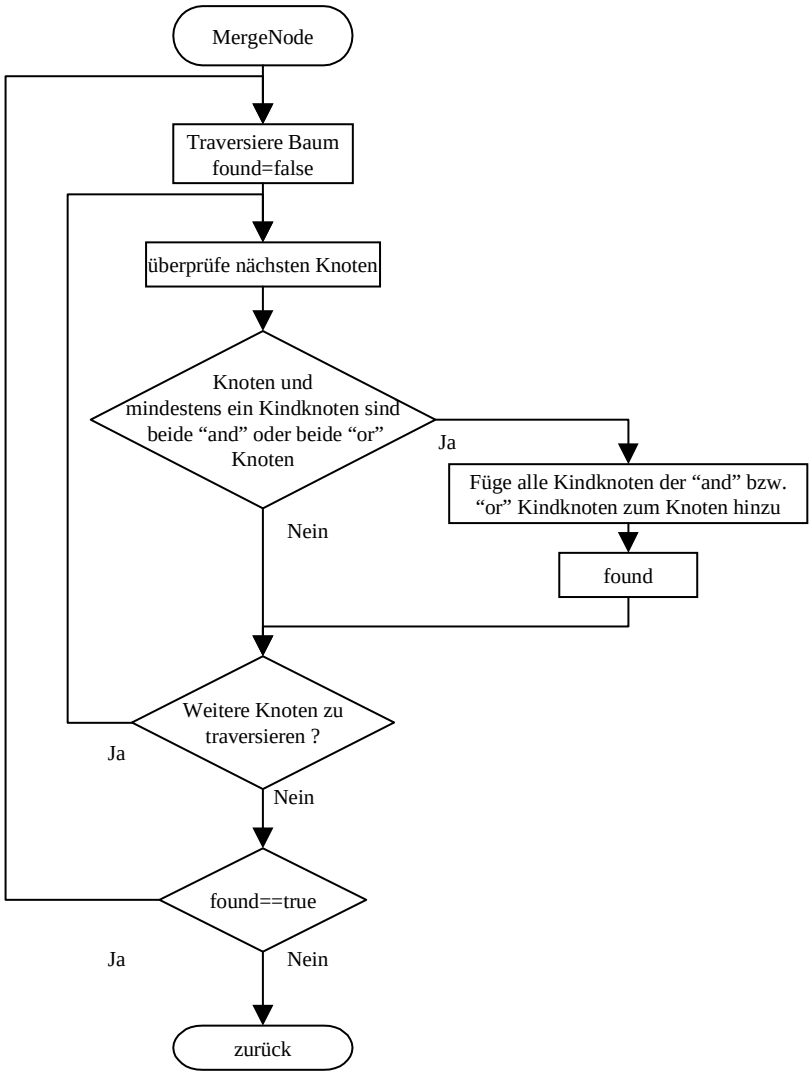


Abb. 28: Flußdiagramm von MergeNode

Nachdem eine *DNF* aufgebaut wurde, kann zum Beispiel noch ein solcher boolscher Ausdruck vorkommen:

(or (or F G) H)

Solche boolsche Ausdrücke sollen zu einer Konjunktion bzw. Disjunktion zusammengefaßt werden. Dazu wird ein Operator implementiert, welcher folgenden Algorithmus ausführt:

Beginnend mit dem Wurzelknoten wird überprüft, ob der aktuelle und dessen Kindknoten ein *and* oder *or* Knoten ist. Ist das der Fall, dann müssen diese beiden Knoten zusammengefaßt werden. Dazu werden alle Kindknoten des gefundenen Kindknotens dem aktuellen Knoten hinzugefügt und der gefundene Kindknoten entfernt. Zum Schluß werden die Referenzen im Baum korrigiert und das Flag *found* gesetzt.

Wurde beim Parsen des Baums das Flag *found* gesetzt, dann wird der Baum erneut geparst.

5.2.4 RemoveTautologies

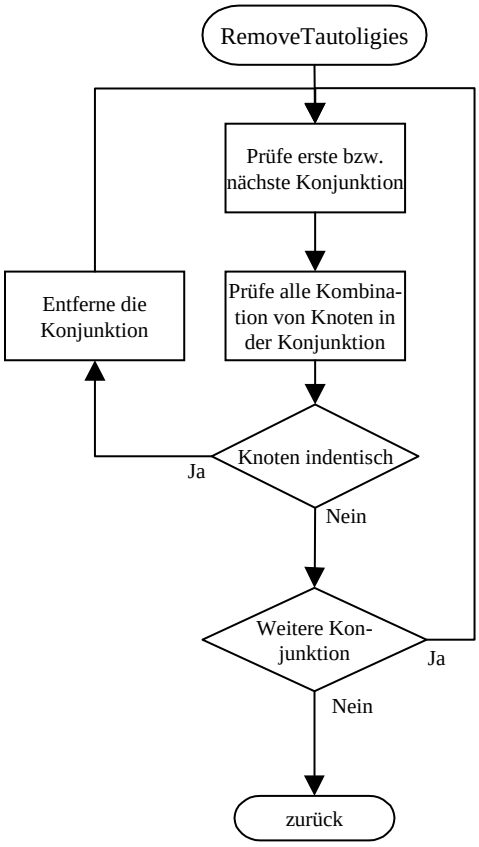


Abb. 29: Flußdiagramm von RemoveTautologies

Nachdem eine *DNF* aufgebaut wurde, kann zum Beispiel noch ein solcher boolscher Ausdruck vorkommen:

(and F (not F) G)

Dazu wird ein Operator implementiert, welcher folgenden Algorithmus ausführt:

Um solche boolsche Ausdrücke zu entfernen, wird unter jedem *and* Knoten überprüft, ob zwei Kindknoten identisch sind, wobei einer der Knoten invertiert ist. Werden solche Knoten gefunden, dann wird der Teilbaum des *and* Knotens entfernt.

Wurden alle Konjunktionen geparst und ist nur noch eine Konjunktion übrig, dann wird der *or* Wurzelknoten entfernt und der verbliebene *and* Knoten ist der neue Wurzelknoten.

Ist keine Konjunktion mehr übrig, dann wird eine Fehlermeldung zurückgegeben.

5.2.5 RemoveDuplicateTerms

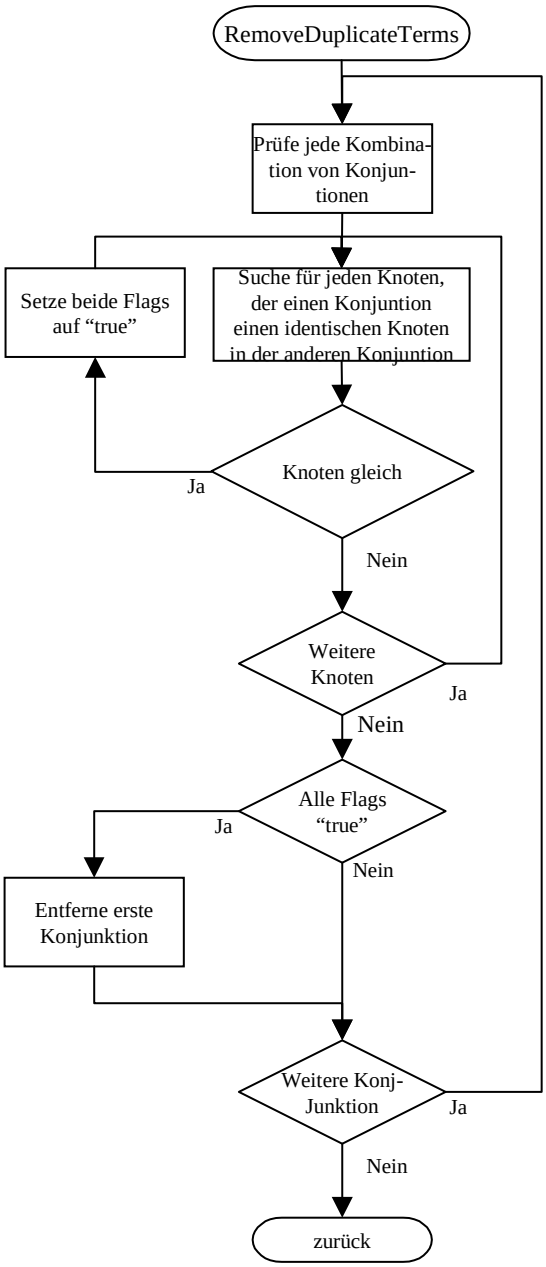


Abb. 30: Flußdiagramm von RemoveDuplicateTerms

Nachdem eine DNF aufgebaut wurde, können zum Beispiel noch solche boolschen Ausdrücke vorkommen:

(or (and A B) (and A B)) *oder* **(or (and A B) (and A B C))**

Solche boolschen Ausdrücke können zu

(and A B)

zusammengefaßt werden. Dazu wird ein Operator implementiert, welcher folgenden Algorithmus ausführt:

Die Konjunktionen müssen miteinander verglichen werden, wenn zumindest zwei Konjunktionen vorhanden sind. Um alle Konjunktionen zu vergleichen, wird beginnend mit der ersten Konjunktion diese mit allen nachfolgenden verglichen.

Für beide Konjunktionen wird für jeden Kindknoten ein Flag angelegt und mit *false* initialisiert.

Es werden alle Knoten der ersten Konjunktion mit allen Knoten der anderen Konjunktion verglichen. Sind die Knoten gleich und die Flags noch nicht *true*, dann werden beide jeweiligen Flags gesetzt.

Wurden alle Kombinationen geprüft, wird für jede Konjunktion überprüft, ob jeweils alle Flags gesetzt sind.

Sind bei der ersten Konjunktion alle Flags gesetzt, dann wird die zweite Konjunktion aus dem Baum entfernt.

Sind bei der ersten Konjunktion nicht alle Flags gesetzt und sind bei der zweiten Konjunktion alle Flags gesetzt, dann wird die erste Konjunktion aus dem Baum entfernt.

Wurden alle Kombinationen von Konjunktion geparst und es ist nur eine Konjunktion übrig, dann wird der *or* Wurzelknoten entfernt und die Konjunktion ist der neue Wurzelknoten.

Ist keine Konjunktion übrig, dann wird eine Fehlermeldung zurückgegeben.

5.2.6 MinimizeConjunctions

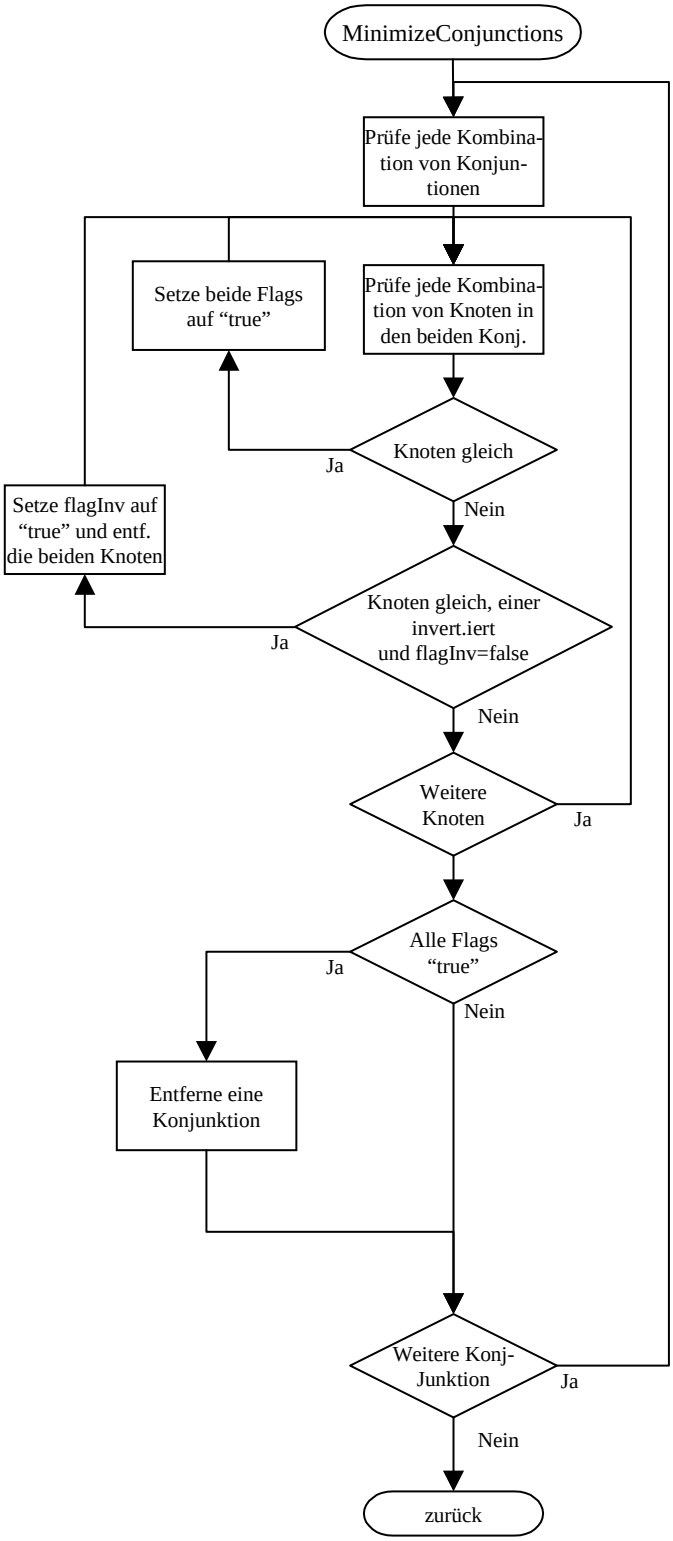


Abb. 31: Flußdiagramm von MinimizeConjunctions

Nachdem eine DNF aufgebaut wurde, kann zum Beispiel noch ein solcher boolscher Ausdruck

vorkommen:

(or (and A B) (and A (not B)))

Ein solcher boolscher Ausdruck kann zu

A

zusammengefaßt werden. Dazu wird ein Operator implementiert, welcher folgenden Algorithmus ausführt:

Die Konjunktionen müssen miteinander verglichen werden, wenn zumindest zwei Konjunktionen vorhanden sind. Dazu wird über alle Paare von Konjunktionen iteriert.

Für beide Konjunktionen wird für jeden Kindknoten ein Flag angelegt und mit *false* initialisiert. Desweiteren wird ein Flag *flagInv* mit *false* initialisiert. Dieses Flag gibt an, ob bereits zu einem Knoten ein identischer aber negierter Knoten gefunden wurde, der entfernt werden soll.

Es werden alle Knoten der ersten Konjunktion mit allen Knoten der anderen Konjunktion verglichen.

Sind die Knoten gleich und die jeweiligen Flags noch nicht *true* (eventuell sind beide invertiert), dann werden beide jeweiligen Flags gesetzt.

Ist einer der Knoten ein *not* Knoten und ist dessen Kindknoten gleich dem Knoten der anderen Konjunktion, sind die jeweiligen Flags noch nicht *true* und ist das *flagInv* Flag noch nicht gesetzt, dann wird der Index des Knotens in der ersten Konjunktion zwischengespeichert und die dazugehörigen Flags und das *flagInv* Flag gesetzt.

Wurden alle Kombinationen geprüft, wird überprüft, ob alle Flags und das *flagInv* Flag gesetzt sind. Ist das der Fall, dann wird die zweite Konjunktion entfernt und der durch den zwischengespeicherten Index indizierte Knoten aus der ersten Konjunktion entfernt.

Wurden alle Kombinationen von Konjunktion geparst und es ist nur eine Konjunktion übrig, dann wird der *or* Wurzelknoten entfernt und die Konjunktion ist der neue Wurzelknoten.

5.2.7 MinimizeConjunctionsSem

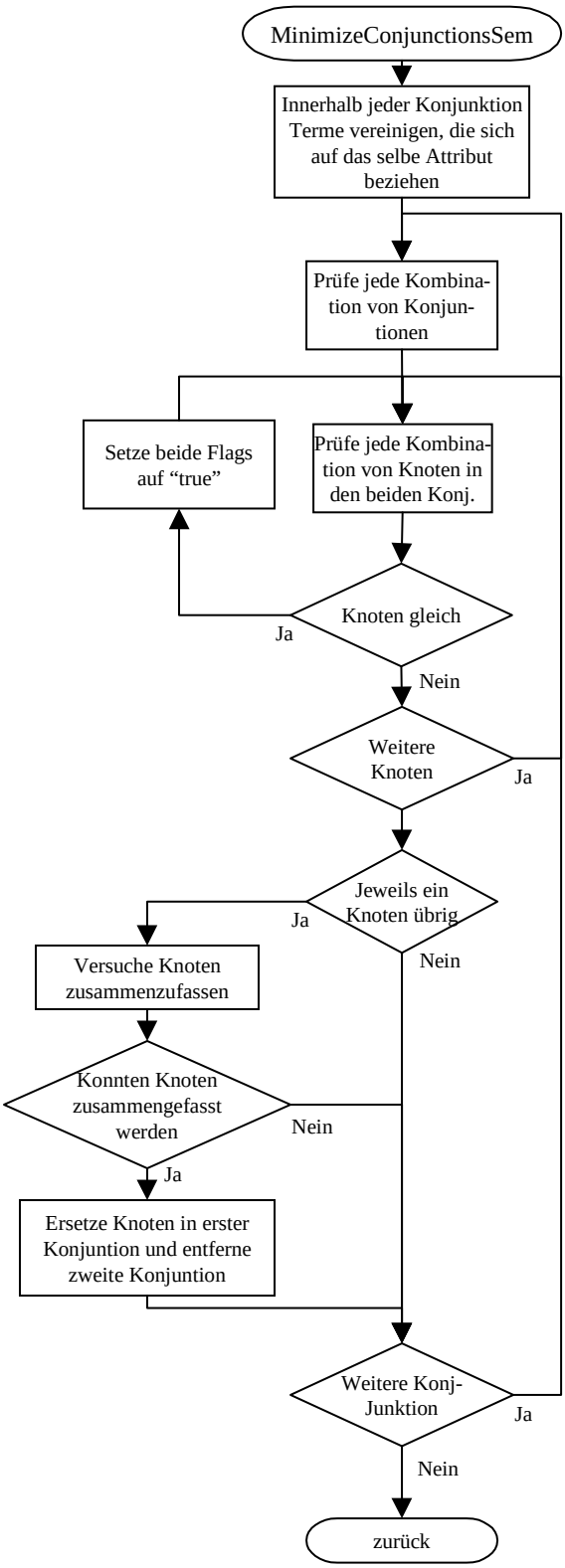


Abb. 32: Flußdiagramm von MinimizeConjunctionsSem

Mit dem Operator M*inimizeConjunctions* konnten dann Konjunktionen zusammengefaßt werden, wenn alle Knoten der beiden Konjunktionen gleich waren, wobei bei einem Paar von

Knoten einer der Knoten invertiert sein mußte. Es gibt aber Fälle, in denen zwei Konjunktionen auch dann zusammengefaßt werden können, wenn alle Knoten bis auf ein Paar gleich sind und die verbleibenden zwei Knoten des Baum semantisch zusammengefaßt werden können.

Dazu sollen folgende Fälle betrachtet werden (die Werte beziehen sich auf das selbe Attribut):

Table 2: Fallunterscheidung beim semantischen Vereinigen von Knoten

Typ des Knotens	direkt/direkt	direkt/inv.	inv./inv.
equal/in Knoten mit Attribut "*type*"	Fall 1	-	Fall 2
inside/overlaps Knoten	Fall 3	-	-Fall 4
bel. equal/in Knoten	Fall 5	-	-

Fall 1:
In diesem Fall können die *type* Werte der *equal* bzw. *in* Knoten in einem *equal* bzw. *in* Knoten zusammengefaßt werden, wobei die Typ-Hierarchie berücksichtigt wird.

Fall 2:
In diesem Fall wird für jeden Typ des ersten Knoten überprüft, welche der Typen des zweiten Knotens Sub-Typ dieses Typs sind. Wurden Sub-Typen gefunden, dann wird der aktuelle Typ im ersten Knoten durch die Sub-Typen ersetzt. Der erste Knoten ist der neue vereinigte Knoten.

Fall 3:
In diesem Fall können die beiden Multi-Polygone zu einem Multipolygon vereinigt werden.

Fall 4:
Auch in diesem Fall können die beiden Multi-Polygone zu einem Multipolygon vereinigt werden. Dabei werden die beiden Multipolygone per intersection zusammengefaßt. Der neue und invertierte Inside Knoten verwendet dieses neue Multipolygon.

Fall 5:
In diesem Fall können die Werte der *equal* bzw. *in* Knoten in einem *equal* bzw. *in* Knoten zusammengefaßt werden.

Der Operator führt dazu folgenden Algorithmus aus:

Es werden alle Konjunktionen geparst. Dabei wird jeweils über alle Elemente der Konjunktion iteriert. Negierte Typ-Vergleiche und direkte Typ-Vergleiche werden jeweils untereinander vereinigt. Die Typen-Vergleiche werden aus der Konjunktion entfernt und durch die vereinigten Typ-Vergleiche ersetzt.

Es werden ebenfalls *overlaps* bzw. *inside* Knoten jeweils vereinigt, solange sie sich auf das selbe Attribut beziehen.

Es werden die Konjunktionen miteinander verglichen, wenn zumindest zwei Konjunktionen vorhanden sind. Um alle Konjunktionen zu vergleichen, wird über alle Paare von Konjunktio-

nen iteriert.

Für beide Konjunktionen wird für jeden Kindknoten ein Flag angelegt und mit *false* initialisiert.

Es werden alle Knoten der ersten Konjunktion mit allen Knoten der anderen Konjunktion verglichen.

Sind die Knoten gleich (eventuell sind beide invertiert), dann werden beide jeweiligen Flags gesetzt.

Wurden alle Kombinationen geprüft, wird überprüft, ob alle Flags bis auf jeweils eines bei jeder Konjunktion gesetzt ist. Ist das der Fall, dann wird überprüft, ob die beiden übrigen Knoten zusammengefaßt werden können.

Dazu wird überprüft, ob einer der fünf Fälle vorliegt. Ist das der Fall, dann wird einer der Knoten aus seiner Konjunktion entfernt und durch den neuen semantisch zusammengefaßten Knoten ersetzt. Die andere Konjunktion wird aus dem Baum entfernt.

Wurden alle Kombinationen von Konjunktion geparst und es ist nur eine Konjunktion übrig, dann wird der *or* Wurzelknoten entfernt und die Konjunktion ist der neue Wurzelknoten.

5.3 Die Operatoren zum Aufbau der RequestMatrix

5.3.1 DetermineServers

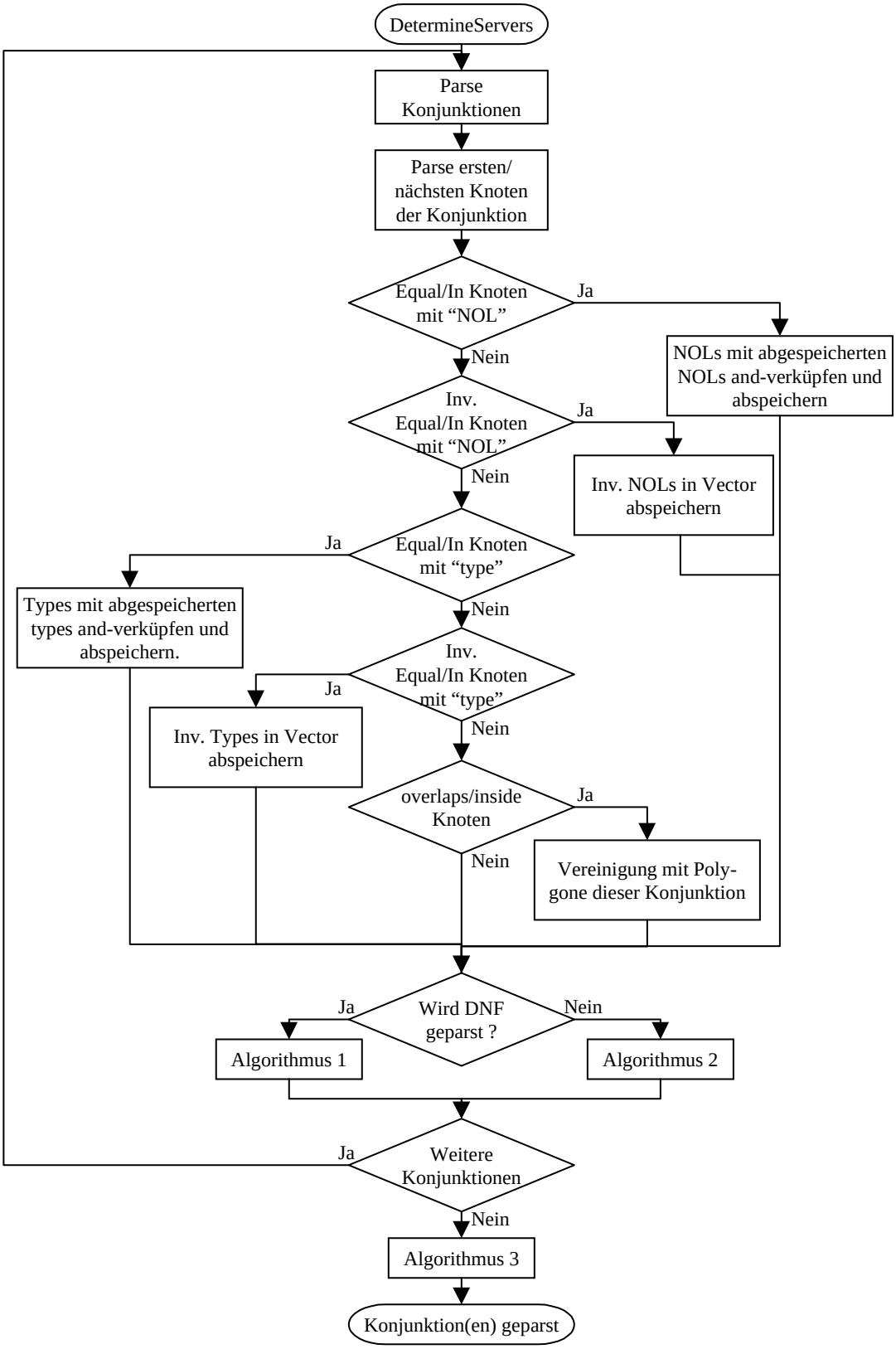


Abb. 33: Flußdiagramm von DetermineServers

Der Operator *DetermineServers* wird dazu verwendet, um beim *ASR* anzufragen, welche Server für eine übergebene Konjunktion oder ein Element des *restriction* Ausdrucks relevant sind. Dem Operator wird eine Referenz auf einen (Teil)-Baum und eine *HashMap* für *Server* Objekte übergeben. Auf diese *HashMap* greifen weitere Instanzen von *DetermineServers* Operatoren nebenläufig zu. Der Zugriff auf die *HashMap* ist deshalb synchronisiert. In einem *Server* Objekt wird für einen Server ein *SpaSeLocator*, die *Capabilities* des Servers (wird noch nicht vom *ASR* unterstützt), und eine Menge von Konjunktionen abgelegt.

Die Methode, welche den Operator startet, führt den folgenden Algorithmus in einem eigenen Thread aus:

Zuerst wird überprüft, ob die Wurzel des übergebenen restriction Baums die Disjunktion einer *DNF* ist. Wurde die Disjunktion übergeben, dann sollen alle Konjunktion geparst werden.

Jeder Kindknoten der Konjunktion bzw. das übergebene Element wird überprüft, ob es ein *equal* oder *in* Knoten ist. Ist das der Fall, dann wird überprüft, ob der Name des Attributes “NOL” oder “type” ist.

Ist der Name des Attributs “NOL” und ist der *equal* bzw. *in* Knoten negiert, dann werden die negierten NOLs zwischengespeichert.

Ist der Name des Attributs “NOL” und ist der *equal* bzw. *in* Knoten nicht negiert, dann wird überprüft, ob bereits ein solcher *equal* oder *in* Knoten mit dem Attribut “NOL” zuvor in dieser Konjunktion geparst wurde.

Ist das nicht der Fall, dann wird eine Liste mit der *NOL* bzw. den *NOLs* angelegt.

Existiert dagegen bereits eine solche Liste, dann wird für jede *NOL* aus dieser Liste überprüft, ob sie gerade im zu parsenden *equal* bzw. *in* Knoten enthalten sind. Wird die jeweilige *NOL* nicht gefunden, dann wird sie aus der Liste entfernt.

Ist der Name des Attributs “type”, dann wird überprüft, ob bereits ein *equal* oder *in* Knoten mit dem Attribut “type” in dieser Konjunktion zuvor geparst wurde.

Ist das nicht der Fall, dann wird eine Liste angelegt, in der *types* abgespeichert werden sollen.

Ansonsten werden die Typen des Knotens in einer temporären Liste abgespeichert. Für jeden Typ aus der Liste werden Sub-Typen in der temporären Liste gesucht. Wurden welche gefunden, dann wird der Typ durch diese ersetzt. Ist der Typ aus der Liste weder ein Sub-Typ einer der Typen der temporären Liste und ist auch keiner der Typen der temporären Liste ein Sub-Typ des Typs aus der Liste, dann wird der Typ aus der Liste entfernt.

Dadurch erhält man Listen mit *NOLs* und *type*s die diesen Ausdruck erfüllen könnten.

Ist der Name des Attributs “type” und ist der Knoten negiert, dann werden die Typen in einer Liste mit negierten Typen zwischengespeichert.

Ist der gerade zu parsende Knoten dagegen ein *inside* oder *overlaps* Knoten und nicht negiert, dann wird das Gebiets-Polygon dieses Knotens mit einem eventuell bereits vorhandenen Multipolygon vereinigt.

Nachdem alle Knoten der Konjunktion geparst wurden, muß unterschieden werden ob eine

DNF geparst wird. Wird eine *DNF* geparst, dann wird Algorithmus 1 ausgeführt. Wird keine *DNF* geparst, dann wird Algorithmus 2 ausgeführt.

Algorithmus 1:

Wurden *NOLs* gefunden, dann werden die nicht invertierten *NOLs* entfernt, welche auch in dem *Vector* mit den invertierten *NOLs* gespeichert sind.

Wurden *NOLs* gefunden, dann wird für jede *NOL* ein Eintrag in der Servers *HashMap* vorgenommen. Als *Restriction* Ausdruck wird die *DNF* verwendet.

Wurden keine NOLs gefunden dann wird überprüft, ob Polygone bzw. Types gefunden wurden:

Wenn sowohl Types als auch Polygone gefunden wurden, dann wird jetzt mit den Types und dem Multipolygon eine Anfrage an das ASR gestellt. Für jeden vom *ASR* zurückgegebenen Server wird unter Verwendung von eventuell vorhandenen negierten Typen dieser Konjunktion überprüft, welche Server Nexus Objekte für diese Konjunktion liefern können: Ist einer der Typen des Servers Sub-Typ der Typen der Konjunktion oder ist einer der Typen der Konjunktion Sub-Typ der Typen des Servers, und ist zumindest ein Typ des Servers kein Sub-Typ der negierten Typen, dann kann der Server Nexus Objekte für diese Konjunktion liefern. Server, welche keine Nexus Objekte liefern können werden wieder entfernt. Für die gefundenen Server werden Einträge in der *HashMap* hinzugefügt.

Wenn nur Types gefunden werden, dann werden die gefundenen Types in einen globalen Vector für alle Konjunktionen kopiert. Danach wird der Vector mit den Types und der Vector mit den negierten Types jeweils in einen Vector eingetragen. Weiter mit Algorithmus 3.

Wenn nur Polygone gefunden wurden, dann wird das Multipolygon mit dem globalen Multipolygon union-verknüpft. Weiter mit Algorithmus 3.

Algorithmus 2:

Wurden *NOLs* gefunden, dann werden die nicht invertierten *NOLs* entfernt, welche auch in dem *Vector* mit den invertierten *NOLs* gespeichert sind.

Wurden *NOLs* gefunden, dann wird für jede *NOL* der *RestrictionOperator* Baum der Konjunktion dupliziert und alle *SimpleCompOperator* Knoten mit *NOL* Vergleichen (auch negierte Knoten) entfernt. Sie werden nicht mehr benötigt. Danach wird der duplizierten Konjunktion ein *equals* Knoten mit der jeweiligen *NOL* hinzugefügt. Diese neue Konjunktion wird dem entsprechenden Objekt in der *Servers HashMap* hinzugefügt, falls ein Objekt für den *SpaSeLocator* der *NOL* bereits existiert. Ansonsten wird ein neues Objekt der *HashMap* hinzugefügt.

Wurden keine *NOLs* gefunden, und wurden Polygone oder Typen gefunden, dann wird die selbe Anfrage an das *ASR* gestellt wie in Algorithmus 1 für den Fall, das Types und Polygone gefunden wurden.

Algorithmus 3:

Wenn eine DNF geparst wurde, dann wird überprüft ob globale Typen oder ein globales Multipolygon gespeichert ist. Dies ist dann der Fall, wenn eine Konjunktion geparst wurde, die nur

Types bzw. nur Polygone enthielt (siehe Algorithmus 1).

Wurden globale Typen gefunden, dann wird eine Anfrage an das ASR gestellt. Dem ASR werden in der Anfrage die globalen Typen, aber keine geographischen Gebiete übergeben. Für die vom ASR zurückgegebenen Server wird für jede Konjunktion überprüft, ob der Server für diese Konjunktion Nexus Objekte zurückgeben kann. Dazu werden der Vektor mit Vektoren der Typen der Konjunktionen und der Vektor mit Vektoren der negierten Typen der Konjunktionen verwendet. Für die Server, welche Nexus Objekte liefern können, wird ein Eintrag in der *Servers HashMap* vorgenommen.

Danach wird überprüft, ob ein globales Multipolygon gespeichert ist. Dies ist dann der Fall, wenn Konjunktionen geparst wurden, welche *overlaps* bzw. *inside* Knoten, aber keine Type bzw. *NOL* Knoten hatten. Dann wird eine Anfrage an das ASR gestellt, wobei nur das Multipolygon übergeben wird. Für jeden gefundenen Server wird ein Eintrag in der Servers HashMap vorgenommen.

Der Thread endet.

Die *get* Methode wartet darauf, daß der Thread endet und gibt dann ein *boolean* Flag zurück, welches angibt ob beim Parsen dieser Konjunktion ein Ausdruck gefunden wurde, der zur Bestimmung von Servern verwendet werden kann.

5.3.2 GetRequestMatrix

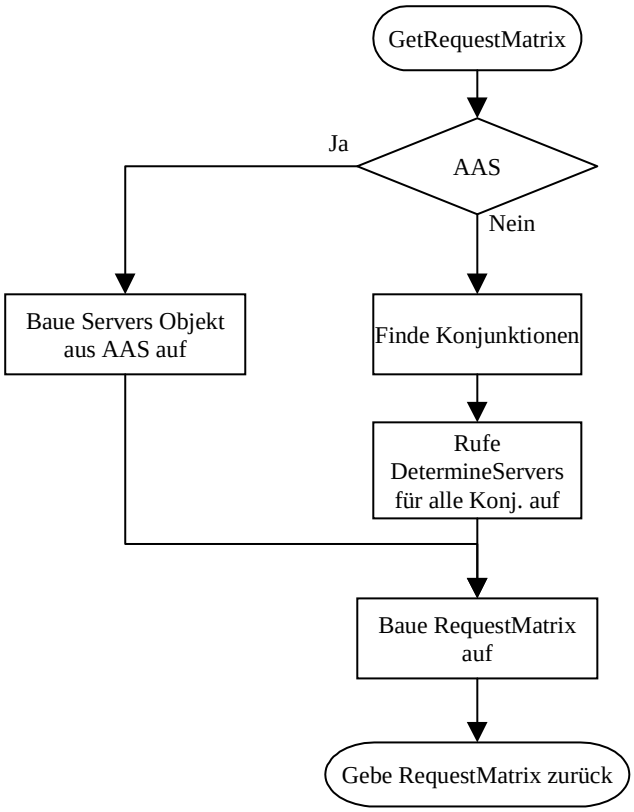


Abb. 34: Flußdiagramm von GetRequestMatrix

Um herauszufinden, an welche Server welche Anfragen, also *AWQL* Dokumente, gesendet werden müssen, muß die *DNF* geparst werden es sei denn, es wurde ein *AAS* Element in der

Anfrage übergeben.

Die *get* Methode führt diesen Algorithmus aus:

Zuerst wird überprüft, ob ein *AAS* Element in der Anfrage vorkommt. Ist das der Fall, dann wird das Parsen der *DNF* und die Anfrage an die *DetermineServers* Operatoren übersprungen und aus dem *AAS* Element eine *HashMap* mit *Servers* Objekten aufgebaut. Den *Servers* Objekten wird der *RestrictionOperator* übergeben, welcher dieser Methode übergeben wurde.

Da in dem AAS Element keine *Capabilities* der Server angegeben werden können, müssen diese beim *ASRQuery* Objekt angefragt werden. Das *ASRQuery* Objekt ist ein Cache für diese Objekte. Kennt das *ASRQuery* Objekt die *Capabilities* noch nicht, dann fragt es zuerst beim *ASR* nach diesen an.

Kommt kein *AAS* Element in der Anfrage vor, dann wir für jede Konjunktion des übergebenen *RestrictionOperator* Objekts ein *DetermineServers* Operator ausgeführt. Wurde der Methode dieses Operators im *RestrictionOperator* ein Element und keine *DNF* übergeben, dann wird dieses Element einem *DetermineServers* Operator übergeben.

Einem *DetermineServers* Operator könnte auch die gesamte *DNF* übergeben werden. In diesem Fall bestimmt der DetermineServers Operator die Server für alle Konjunktionen und es werden eventuell weniger Anfragen an das *Area Service Register* gestellt. Allerdings wird dann an jeden Server die komplette *DNF* übergeben.

Nachdem die *DetermineServers* Operatoren die Ergebnisse in der *Servers HashMap* zurückgegeben haben, kann die *RequestMatrix* aufgebaut werden.

Dazu wird aus jedem *Servers* Objekt und dem *AWQLQuery* Objekt ein neues *Request* Objekt aufgebaut und in der *RequestMatrix* eingetragen. Das *Servers* Objekt enthält den *SpaSeLocator* des Servers, an den die Anfrage zu stellen ist und den *RestrictionOperator* Baum, der von diesem Server zu beantwortenden Konjunktionen. Das *RestrictionOperator* Objekt des *AWQLQuery* Objekts ist das Objekt, welches vor der Konvertierung des *Restriction*-Ausdrucks die Wurzel des *Restriction*-Ausdrucks war. Dieses Objekt darf nicht verwendet werden. Ein Ersetzen dieses Objekts im *AWQLQuery* Objekt durch das *RestrictionOperator* Objekt des *Servers* Objekt würde es notwendig machen, für jeden Request eine Kopie dieses Objekts zu erstellen.

Die *RequestMatrix* wird zurückgegeben.

5.3.3 CacheGetRequestMatrix

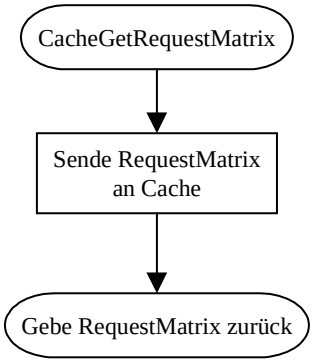


Abb. 35: Flußdiagramm von CacheGetRequestMatrix

Um den Cache einzubinden, wird die *RequestMatrix* mit diesem Operator dem Cache übergeben. Der Cache gibt eine geänderte *RequestMatrix* zurück. Der Cache taucht nun selbst als Server in der *RequestMatrix* auf. Der Operator gibt diese *RequestMatrix* zurück wird.

Die *get* Methode, die das *RequestMatrix* Objekt zurückgibt, holt zuerst ein *RequestMatrix* Objekt vom *getRequestMatrix* Operator. Danach übergibt es dieses Objekt dem Cache und gibt die Antwort des Cache zurück.

5.4 Die Operatoren zur Anfrage von Servern und Bearbeiten der Antwort

5.4.1 ServerQuery

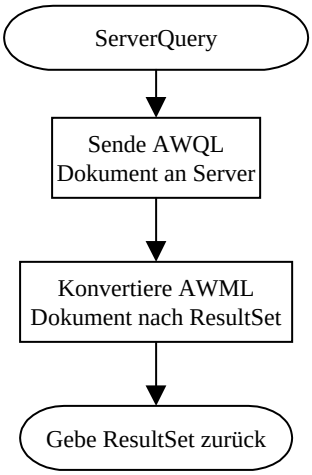


Abb. 36: Flußdiagramm von ServerQuery

Um Anfragen an einen Server des Nexus Systems zu senden, wie zum Beispiel einen *Spatial Model Server* oder den *Location Service*, oder um eine Anfrage an den *Cache* zu stellen wird der Operator *ServerQuery* verwendet. Dieser Operator sendet eine Anfrage an die Methode

query.

Dem Operator wird ein *Request* und ein *ServerInterface* Objekt übergeben.

Aus diesem Objekt wird das *AWQLQuery* Objekt und der *SpaSeLocator* des Servers ausgelesen. Um das *AWQL* Dokument aufzubauen, wird das *RestrictionOperator* Objekt des *Server* Objekts verwendet.

Der Operator startet den Algorithmus in einem eigenen Thread.

Zuerst übergibt der Thread das *Request* Objekt der *query* Methode des *ServerInterface* Objekts. Dies ist entweder der *Cache* oder das Objekt, welches den Request an einen Nexus Server per *ApacheSOAP* übergibt. Das zurückgegebene *ResultSet* wird abgespeichert.

Wurde der Methode *setNotifyOperator* ein *MergeResultSet* Operator übergeben, dann wird dessen *notifyOperator* Methode aufgerufen. Der Thread endet.

Die *get* Methode, die das *ResultSet* zurückgibt überprüft, ob der Klasse ein *notifyOperator* übergeben wurde. Wurde einer übergeben, dann wartet die Methode, bis der Thread endet. Danach wird das *ResultSet* zurückgegeben. Liegt noch keine Antwort des Servers vor, dann ist dies eine *null* Referenz.

5.4.2 CacheInsert

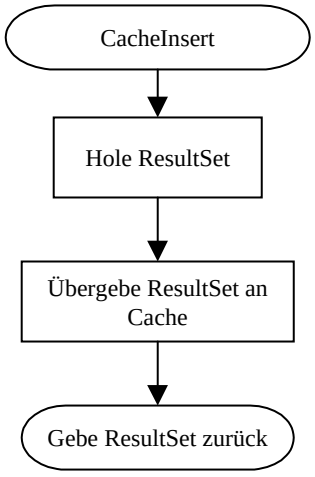


Abb. 37: Flußdiagramm von CacheInsert

Nachdem ein *ServerQuery* Operator ein *ResultSet* von einem Nexus Server erhalten hat, müssen diese Nexus-Objekte und der Request, welcher dem Nexus Server übergeben wurde, dem Cache übergeben werden. Dazu wird dieser Operator verwendet.

Dem Operator wird unter anderen ein *Request* Objekt und das *Cache* Objekt übergeben.

Der Operator führt den Algorithmus in einem eigenen Thread aus.

Die *get* Methode welche ein *ResultSet* zurückgeben soll holt zuerst das *ResultSet* von seinem

Kind-Operator. Hat der Kindoperator ein *ResultSet* zurückgegeben, dann wird es kopiert und der Thread aufgeweckt. Der Thread wiederum übergibt das kopierte *ResultSet* dem *Cache*. Dem *Cache* wird eine Kopie übergeben, da der *ApplyRestriction* Operator und der *Cache* sonst zeitgleich versuchen könnten, auf das *ResultSet* zuzugreifen und der *ApplyRestriction* Operator das *ResultSet* verändern kann.

Die *get* Methode gibt das *ResultSet* zurück.

5.4.3 ApplyRestriction

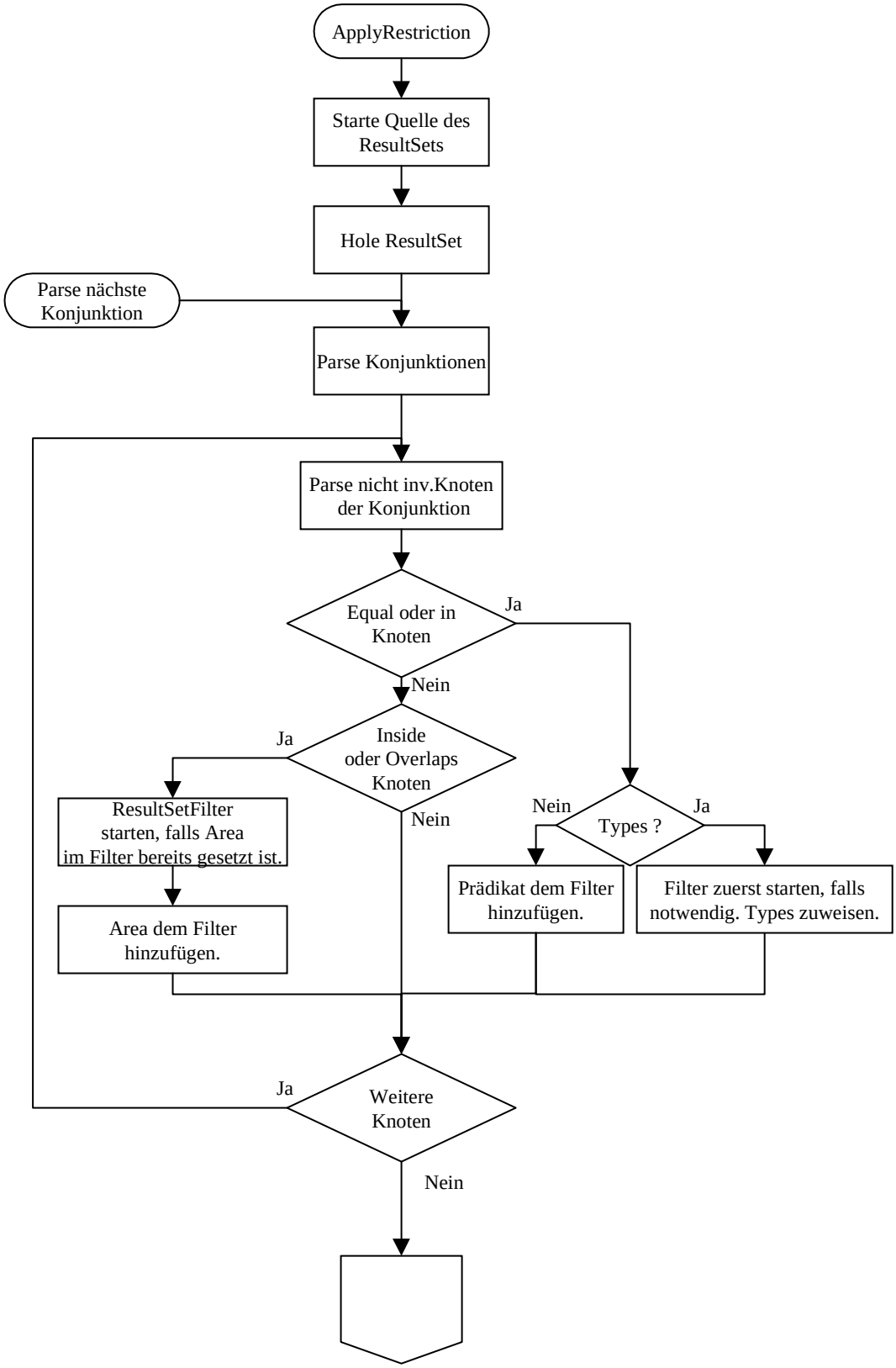


Abb. 38: Flußdiagramm von ApplyRestriction

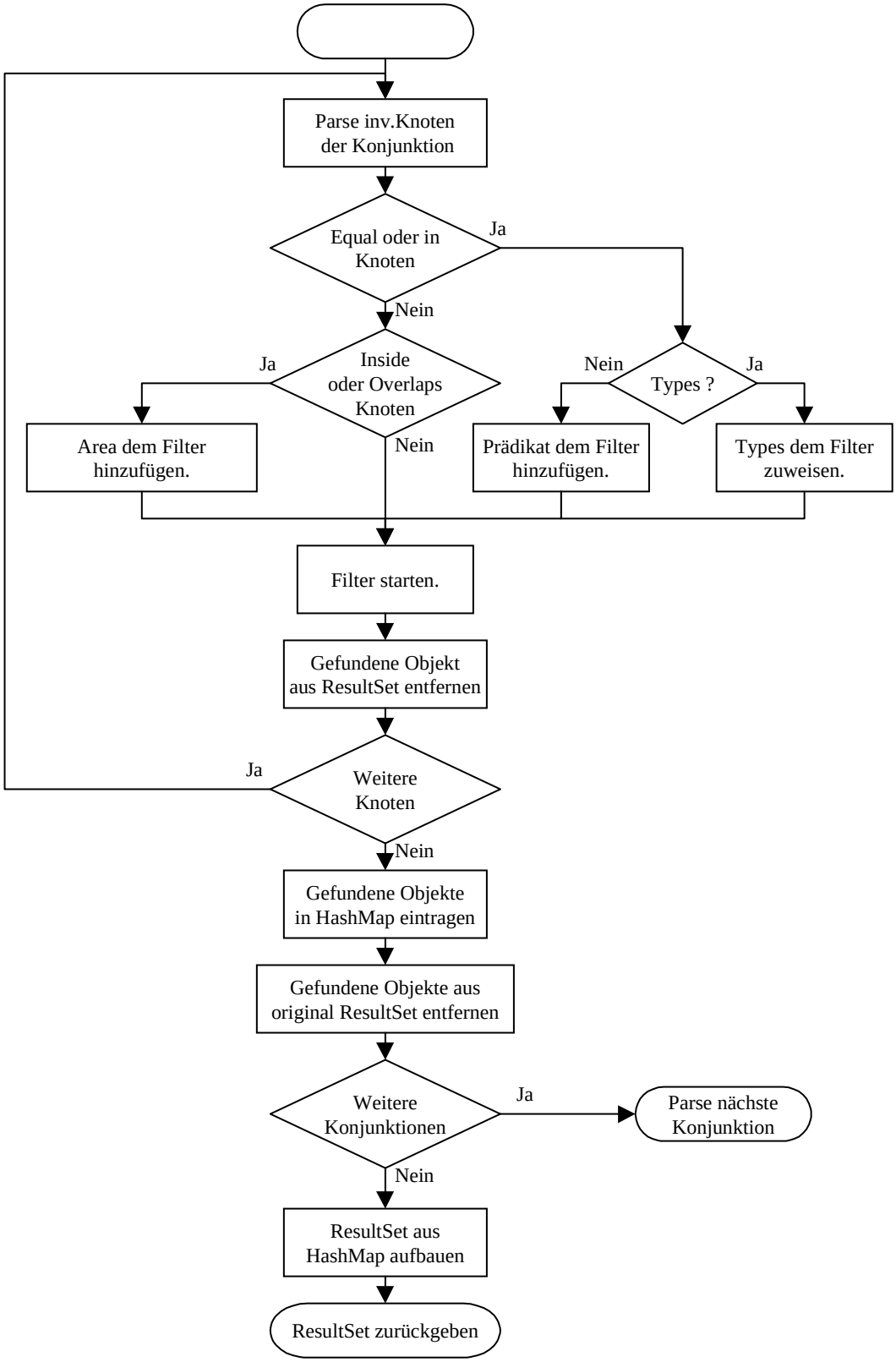


Abb. 39: Flußdiagramm von ApplyRestriction

Unter Umständen können nicht alle Server der Nexus Plattform einen *restriction* Ausdruck parsen, wie zum Beispiel der *ContextCube,* und geben deshalb alle Objekte unabhängig vom übergebenen *restriction* Ausdruck zurück. In diesem Fall muß in den Anfragebaum dieser *ApplyRestriction* Operator eingefügt werden.

Die Methode des Operators, die den Operator startet, ruft die entsprechende Methode des Kind-Operators auf und kehrt dann zurück.

Die *get* Methode des Operators, die das *ResultSet* zurückgibt, holt zuerst das *ResultSet* vom Kind-Operator. Wurde ein *ResultSet* zurückgegeben, dann muß der Restriction-Ausdruck auf das *ResultSet* angewendet werden. Dazu wird dieser Algorithmus ausgeführt:

Jede Konjunktion wird geparst. Beim Parsen jeder Konjunktion werden zuerst die nicht negierten Knoten geparst. Je nach Typ des Knoten werden die boolschen Bedingungen zu einem *ResultSetFilter* Objekt hinzugefügt. Falls die Bedingung nicht mehr hinzugefügt werden kann, wird zuerst gefiltert, indem das *ResultSetFilter* Objekt der *query* Methode des *ResultSets* übergeben wird. Das neue *ResultSet* wird beim Parsen dieser Konjunktion weiterverwendet.

Danach werden beim Parsen der Konjunktion alle negierten Knoten geparst. Hier wird nach jedem Knoten gefiltert. Die gefundenen Nexus Objekte werden aus dem verwendeten *ResultSet* entfernt. Es ist notwendig nach jedem Knoten zu filtern, da es für einen Ausdruck wie **(and a (not b) (not c))=>(and a (not (or b c)))** nicht möglich ist, die oder-Verknüpfung im Filter Objekt zu repräsentieren

Nachdem alle Knoten der Konjunktion geparst wurden, sind in dem verwendeten *ResultSet* nur noch Nexus Objekte gespeichert, welche die gerade geparste Konjunktion erfüllen. Diese werden in eine *ArrayList* eingetragen. Aus dem ursprünglich vom Kind-Operator zurückgegebenen *ResultSet* werden diese Nexus Objekte entfernt, da sie in den anderen Konjunktionen nicht mehr überprüft werden müssen.

Nachdem alle Konjunktionen geparst wurden, werden die Nexus Objekte der *HashMap* in ein neues *ResultSet* eingetragen, welches zurückgegeben werden wird.

5.4.4 SortResultSet

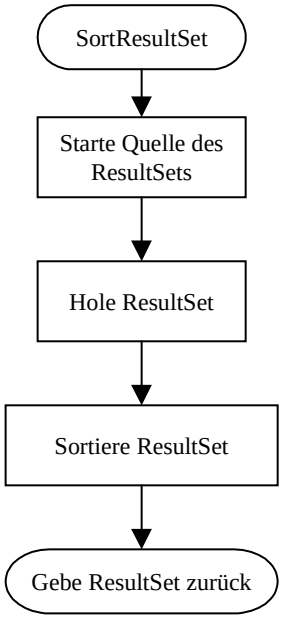


Abb. 40: Flußdiagramm von SortResultSet

Wenn ein MergeResultSet Operator sortierte *ResultSets* schneller mischen kann, dann kann dieser Operator vor den *MergeResultSet* Operator dann eingefügt werden, wenn der Nexus Server einer Quelle nicht dazu in der Lage ist, die Nexus Objekte nach der *ObjectId* sortiert zurückzugeben.

Dazu soll in einem ersten Schritt zuerst das *ResultSets* sortiert werden. Es wurde ein *SortComparator* Objekt implementiert, welches das *Comparator* Interface implementiert. Dieses verwendet die *ObjectId* als Vergleichskriterium. Dadurch wird das *ResultSet* nach der *ObjectId* sortiert. Durch die *ObjectId* wird ein Objekt eindeutig identifiziert.

Dem Operator wird ein Kindknoten des Anfragebaums, also ein *ServerQuery* Operator, und das *Comparator* Objekt übergeben.

Die *start* Methode des Operators, die den Operator startet, ruft die entsprechende Methode des Kind-Operators auf und kehrt dann zurück.

Die Methode des Operators, die das *ResultSet* zurückgibt, holt zuerst das *ResultSet* vom Kind-Operator. Wurde ein *ResultSet* zurückgegeben, dann wird es sortiert. Danach wird das *ResultSet* zurückgegeben.

5.4.5 MergeResultSet

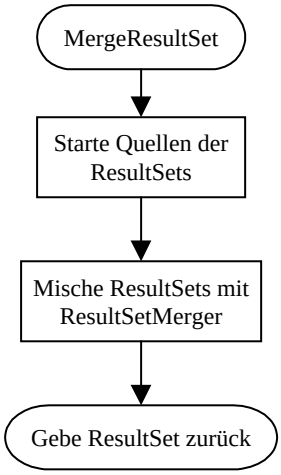


Abb. 41: Flußdiagramm von MergeResultSet

Um die Objekte der einzelnen Server in einem *ResultSet* zu vereinigen, werden *ResultSets* vereinigt sobald ein weiterer Kind-Operator dieses Operators ein *ResultSet* zurückgeben kann bis alle Kind-Operatoren entweder eine *SOAPException* oder ihr *ResultSet* zurückgegeben haben.

Die *start* Methode des Operators, welche den Operator startet ruft die entsprechenden Methoden der Kind-Operatoren auf und kehrt dann zurück.

Der Algorithmus dieses Operators wird in einem Thread ausgeführt. Der Thread wird vom Konstruktor gestartet.

Die Methode des Operators, welche das *ResultSet* zurückgibt, wartet bis der Thread dieses Operators beendet ist und gibt dann das *ResultSet* zurück.

Der Thread wartet, bis er von einem der Kind-Operatoren aufgeweckt wird. Wurde er aufgeweckt, dann holt er von dem Kind-Operator welcher ihn aufgeweckt hat das *ResultSet*. Holt der Thread das erste mal ein *ResultSet*, dann wird es zwischengespeichert. Ansonsten wird es mit dem gespeicherten vereinigt. Dazu wird der schon vorhandene *ResultSetMerger* eingesetzt. Wurden die *ResultSets* aller Kind-Operatoren, welche keine Fehlermeldung zurückgegeben haben vereinigt, dann endet der Thread.

Mobile Objekte besitzen ein Attribut, welches den Zeitpunkt angibt, zu dem die Position des mobilen Objekts zuletzt bestimmt wurde. Kommt ein mobiles Objekt in mehr als einem *ResultSet* vor, dann muß bei dem Vereinigen der Objekte darauf geachtet werden, daß die aktuelleren Attribute übernommen werden. Da dies von *ResultSetMerger* noch nicht unterstützt wird, sollte diese Klasse dementsprechend erweitert werden.

5.4.6 ServerUpdate

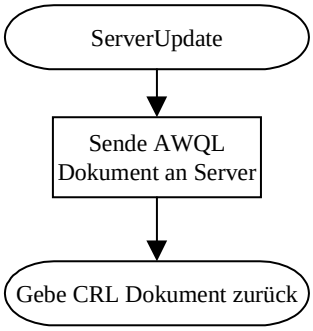


Abb. 42: Flußdiagramm von ServerUpdate

Um Updates an einen Server des Nexus Systems zu senden, wie zum Beispiel einen *Spatial Model Server* oder den *Location Service*, oder um Updates an den *Cache* zu übergeben, wird der Operator *ServerUpdate* verwendet. Dieser Operator sendet eine Anfrage an die Methode *update*. Die *notifyOperator* und *setNotifyOperator* Methoden werden bei diesem Operator in der selben weise verwendet, wie bei den anderen äquivalenten Query-Operatoren.

Dem Operator wird ein *Request* und ein *ServerInterface* Objekt übergeben.

Der Operator startet den Algorithmus in einem eigenen Thread.

Zuerst übergibt der Thread das *Request* Objekt der *update* Methode des *ServerInterface* Objekts. Dies ist entweder der *Cache* oder das Objekt, welches den Request an einen Nexus Server per *ApacheSOAP* übergibt. Nachdem der Thread die Antwort erhalten und abgespeichert hat, endet er.

Die *get* Methode, welche das *CRLSet* Objekt zurückgibt wartet darauf, daß der Thread endet und gibt das *CRLSet* Objekt zurück.

5.4.7 ServerInsert

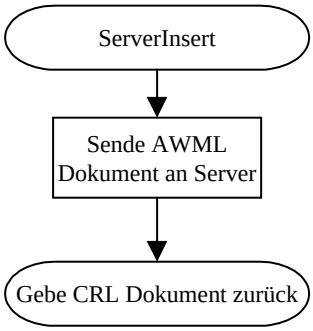


Abb. 43: Flußdiagramm von ServerInsert

Um Einfüge-Aufträge an einen Server des Nexus Systems zu senden, wie zum Beispiel einen

Spatial Model Server oder den *Location Service*, oder um Nexus Objekte im *Cache* zu registrieren, wird der Operator *ServerInsert* verwendet. Dieser Operator sendet eine Anfrage an die Methode *insert*. Die *notifyOperator* und *setNotifyOperator* Methoden werden bei diesem Operator in der selben weise verwendet, wie bei den anderen äquivalenten Query-Operatoren. Tritt beim Registrieren von Objekten ein Fehler auf, also wenn zum Beispiel der Server nicht angesprochen werden kann, dann wird ein *CRLSet* zurückgegeben, in der das Ergebnis für alle *NOLs* false ist.

Dem Operator wird ein *SpaSeLocatorKey*, ein *ResultSet* und ein *ServerInterface* Objekt übergeben.

Der Operator startet den Algorithmus in einem eigenen Thread.

Ist der *SpaSeLocatorKey null*, dann wird das *ResultSet* and die *insert* Methode des *ServerInterface* Objekts übergeben. In diesem Fall ist das *ServerInterface* Objekt der *Cache*.

Ist der *SpaSeLocatorKey* nicht *null*, dann wird der *SpaSeLocatorKey* in ein *Server* Objekt und dieses in ein *Request* Objekt abgespeichert. Das *Request* Objekt und das *ResultSet* werden der *insert* Methode des *ServerInterface* Objekts übergeben. Dieses Objekt übergibt das ResultSet dem Nexus Server und gibt die Antwort zurück.

Nachdem der Thread die Antwort erhalten und abgespeichert hat, endet er.

Die *get* Methode, welche das *CRL* Objekt zurückgibt wartet darauf, daß der Thread endet und gibt das *CRL* Objekt zurück.

5.4.8 ServerDelete

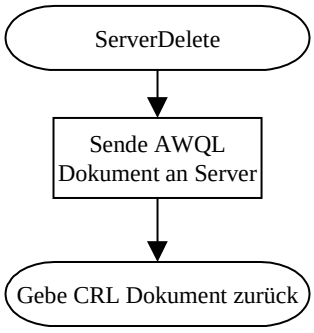


Abb. 44: Flußdiagramm von ServerDelete

Um Lösch-Aufträge an einen Server des Nexus Systems zu senden, wie zum Beispiel einen *Spatial Model Server* oder den *Location Service*, oder um Nexus Objekte aus dem *Cache* zu entfernen, wird der Operator *ServerDelete* verwendet. Dieser Operator übergibt eine Anfrage an die Methode *delete*. Die *notifyOperator* und *setNotifyOperator* Methoden werden bei diesem Operator in der selben weise verwendet, wie bei den anderen äquivalenten Query-Operatoren.

Dazu führt dieser Operator den selben Algorithmus aus, wie der *ServerUpdate* Operator. Allerdings übergibt dieser Operator die Anfrage an die Methode *delete*.

5.4.9 MergeCRL

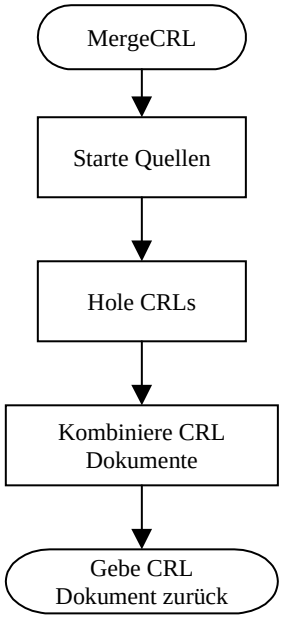


Abb. 45: Flußdiagramm von MergeCRL

Dieser Operator wird von den Methoden *update*, *insert* und *delete* dazu verwendet, um die Antworten der Server, also *CRL* Objekte, zu einem *CRL* Objekt zusammenzufassen. Die *notifyOperator* Methode werden bei diesem Operator in der selben Weise verwendet, wie bei dem äquivalenten *MergeResultSet*-Operator.

Die *start* Methode des Operators ruft die start Methoden der Kindoperatoren.

Die *get* Methode des Operators wartet darauf, bis der vom Konstruktor gestartete Thread abgearbeitet wurde und gibt dann das *CRLSet* Objekt zurück.

Der Thread vereinigt die *CRLSet* Objekte der Kindoperatoren so wie der *MergeResultSet* Operator die *ResultSets* vereinigt.

6 Feinentwurf

6.1 Die Föderation

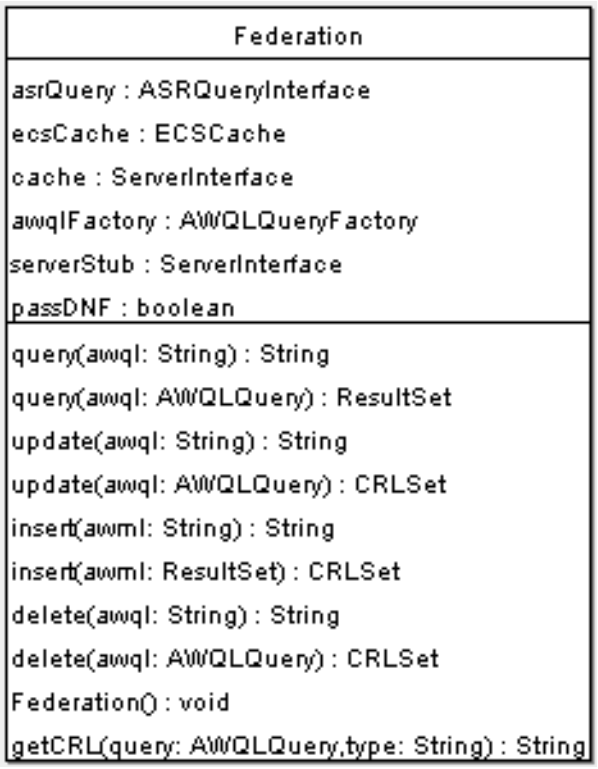


Abb. 46: Klassendiagram der Föderation

6.1.1 Federation

query
Die Methode bearbeitet eine Anfrage per *AWQL*. Dazu wird die Methode *query* benutzt. Es wird ein *AWML* Dokument zurückgegeben.

query
Die Methode gibt Nexus Objekte zurück, welche das übergebene *AWQLQuery* Objekt erfüllen. Die Methode verwendet *DNF* Operatoren, *RequestMatrix* Operatoren und Query Operatoren. Die Methode gibt die Nexus Objekte in einem *ResultSet* zurück.

update
Die Methode konvertiert das übergebene *AWQL* Dokument in ein *AWQLQuery* Objekt, ruft die untere *update* Methode auf, konvertiert das erhaltene *CRLSet* Objekt in ein *CRL* Dokument und gibt dieses zurück.

update
Die Methode aktualisiert Attribute von Nexus Objekte. Die Methode verwendet die Methode *getCRL*. Es wird ein *CRLSet* Objekt zurückgegeben.

insert
Die Methode konvertiert das übergebene *AWML* Dokument in ein *ResultSet*, ruft die untere

insert Methode auf, konvertiert das erhaltene *CRLSet* Objekt in ein *CRL* Dokument und gibt dieses zurück.

insert
Die Methode registriert neue Nexus Objekte bei den Servern. Die Methode verwendet *QueryCRL* Operatoren und Query Operatoren. Es wird ein *CRLSet* Objekt zurückgegeben.

delete
Die Methode konvertiert das übergebene *AWQL* Dokument in ein *AWQLQuery* Objekt, ruft die untere *delete* Methode auf, konvertiert das erhaltene *CRLSet* Objekt in ein *CRL* Dokument und gibt dieses zurück.

delete
Die Methode entfernt Nexus Objekte auf den Servern. Die Methode verwendet die Methode *getCRL*. Es wird ein *CRLSet* Objekt zurückgegeben.

getCRL
Die Methode aktualisiert entweder Attribute von Nexus Objekten oder entfernt die Nexus Objekte auf den Servern. Die Methode verwendet DNF Operatoren, *RequestMatrix* Operatoren und *QueryCRL* Operatoren. Es wird ein *CRL* Dokument zurückgegeben.

6.2 Die DNF Operatoren

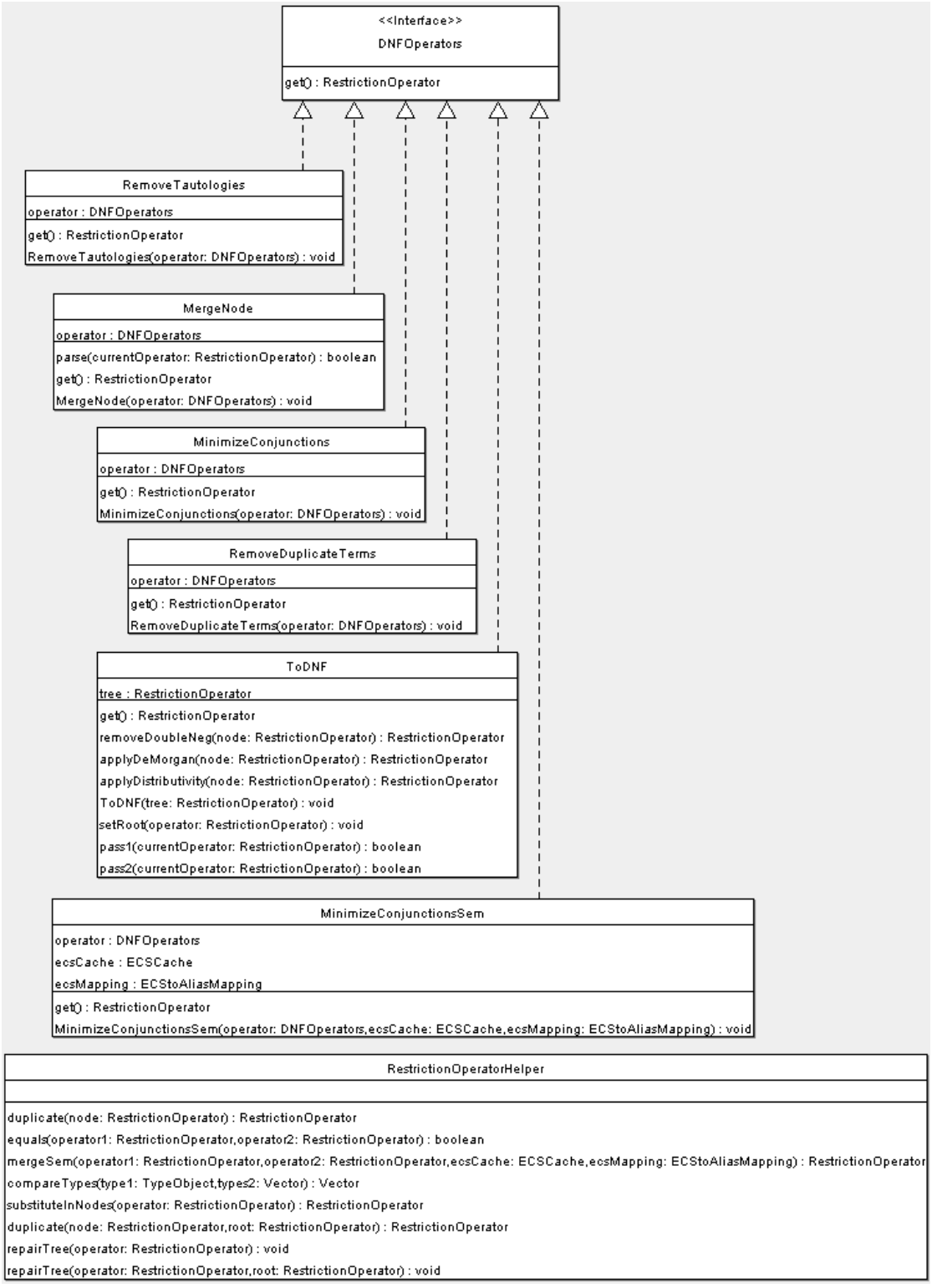


Abb. 47: Klassendiagram der DNFOperator Klassen

6.2.1 DNFOperators

get
Diese Methode der DNF Operatoren holt den *RestrictionOperator* des Kindknotens, bearbeitet ihn, und gibt ihn zurück.

6.2.2 MergeNode

get
Die Methode ruf die *get* Methode des Kindknotens auf und ruft dann die Methode *parse* für den *RestrictionOperator* des Wurzelknotens auf, bis keine Änderungen im Baum mehr ausgeführt wurden. Der bearbeitete Baum*s* wird zurückgegeben.

parse
Die Methode fasst einen or (oder and) Knoten und einen or (bzw. and) Kindknoten zu einem Knoten zusammen. Für alle eventuell vorhandenen Kindknoten wird die Methode *parse* rekursiv aufgerufen.

6.2.3 RemoveTautologies

get
Die Methode sucht in den Konjunktionen nach gleichen Operatoren die sowohl Invertiert als auch nichtinvertiert vorkommen. Solche Konjunktionen können entfernt werden. Die Methode holt den *RestrictionOperator* des Wurzelknotens des Restriction Ausdrucks vom Kindoperator und gibt den *RestrictionOperator* des bearbeiteten Baum*s* zurück.

6.2.4 RemoveDuplicateTerms

get
Die Methode sucht nach identischen Konjunktionen in einer DNF. Wurde zwei identische Konjunktion gefunden, dann kann eine davon entfernt werden. Die Methode holt den *RestrictionOperator* des Wurzelknotens des Restriction Ausdrucks vom Kindoperator und gibt den *RestrictionOperator* des bearbeiteten Baum*s* zurück.

6.2.5 MinimizeConjunctions

get
Sind zwei Konjunktionen identisch bis auf jeweils einen Knoten und sind diese beiden Knoten gleich, wobei allerdings einer invertiert ist, dann können die beiden Konjunktionen zu einer zusammengefaßt werden. Die Methode holt den *RestrictionOperator* des Wurzelknotens des Restriction Ausdrucks vom Kindoperator und gibt den *RestrictionOperator* des bearbeiteten Baum*s* zurück.

MinimizeConjunctionsSem

get
So wie auch die Methode *get* der Klasse *MinimizeConjunctions* fast diese Methode Konjunktionen zusammen. Allerdings berücksichtigt diese Methode die Semantik der Elemente des Restriction Ausdrucks. Die Methode holt den *RestrictionOperator* des Wurzelknotens des Restriction Ausdrucks vom Kindoperator und gibt den *RestrictionOperator* des bearbeiteten Baum*s* zurück.

6.2.6 ToDNF

ToDNF
Dem Konstruktor wird der noch unbearbeitete Restriction Ausdruck des *AWQL* Dokuments übergeben. Der *RestrictionOperator* ist der Wurzelknoten des Ausdrucks.

get
Die Methode holt den *RestrictionOperator* vom Kindknoten, indem dessen Methode *get* aufgerufen wird. Danach wird zuerst die Methode pass1 aufgerufen, bis keine Änderung im Baum mehr vorgenommen wurde. Dann wird die Methode pass2 aufgerufen, bis keine Änderung im Baum mehr vorgenommen wurde. Der *RestrictionOperator* wird zurückgegeben.

pass1
Die Methode ruft für den übergebenen *RestrictionOperator* gegebenenfalls die Methoden *removeDoubleNeg* bzw. *applyDeMorgan* auf. Danach wird für jeden Kindknoten die Methode *pass1* rekursiv aufgerufen.

pass2
Die Methode ruft für den übergebenen *RestrictionOperator* gegebenenfalls die Methode *applyDistributivity* auf. Danach wird für jeden Kindknoten die Methode *pass2* rekursiv aufgerufen.

removeDoubleNeg
Die Methode wird von der *get* Methode dazu verwendet, um Doppelnegationen aus dem Baum zu entfernen.

applyDeMorgan
Die Methode wird von der *get* Methode dazu verwendet, um die deMorgansche Regel anzuwenden.

applyDistributivity
Die Methode wird von der *get* Methode dazu verwendet, um die Distributivitäts Regel anzuwenden.

setRoot
Diese Methode parst den *RestrictionOperator* Baum rekursiv und korrigiert alle Referenzen auf den Wurzelknoten des *RestrictionOperator* Baums.

6.2.7 RestrictionOperatorHelper

duplicate
Die Methode erstellt eine Kopie des übergebenen *RestrictionOperators* und seiner Kindknoten und gibt ihn zurück. Die Methode arbeitet rekursiv.

equals
Die Methode gibt *true* zurück, wenn beide übergebenen Restriction Operatoren *SimpleCompOperator* oder *SpatialCompOperator* Objekte und gleich sind.

mergeSem
Die Methode versucht die zwei übergebenen Operatoren semantisch oder zu verknüpfen. Können die beiden zu einem Operator zusammengefaßt werden, dann wird dieser zurückgegeben.

Ansonsten wird eine null Referenz zurückgegeben.

compareTypes
Diese Methode vergleich einen Nexus Type mit einer Liste von Nexus Types. Sind Types in type2 Subtypen von type1, dann werden diese in einem Vektor zurückgegeben. Ist keiner der Typen in type2 ein Subtyp von type1 und ist type1 kein Subtyp von einem der Types in type2, dann wird ein leerer Vektor zurückgegeben. Ist type1 ein Subtyp eines der Types in type2, dann wird eine *null* Referenz zurückgegeben. Ist type1 und einer der Types in type2 gleich, dann wird das Objekt, welches in type1 übergeben wurde in einem Vektor (mit nur diesem Objekt) zurückgegeben.

substituteInNodes
Die Methode ersetzt rekursiv alle in einem *RestrictionOperator* Baum vorkommenden *in*-Knoten durch mehrere *equal*-Knoten. Die *equal*-Knoten sind dann Kindknoten eines neuen *or*-Knotens. Dadurch wird ein *AWQL* Ausdruck in ein *AWQL 2.0* Ausdruck konvertiert.

repairTree
Die Methode parst den übergebenen Baum rekursiv und korrigiert bzw. setzt die Referenzen im Baum, also die Referenzen auf den Superoperator und den Root-Operator.

6.3 Die RequestMatrix Operatoren

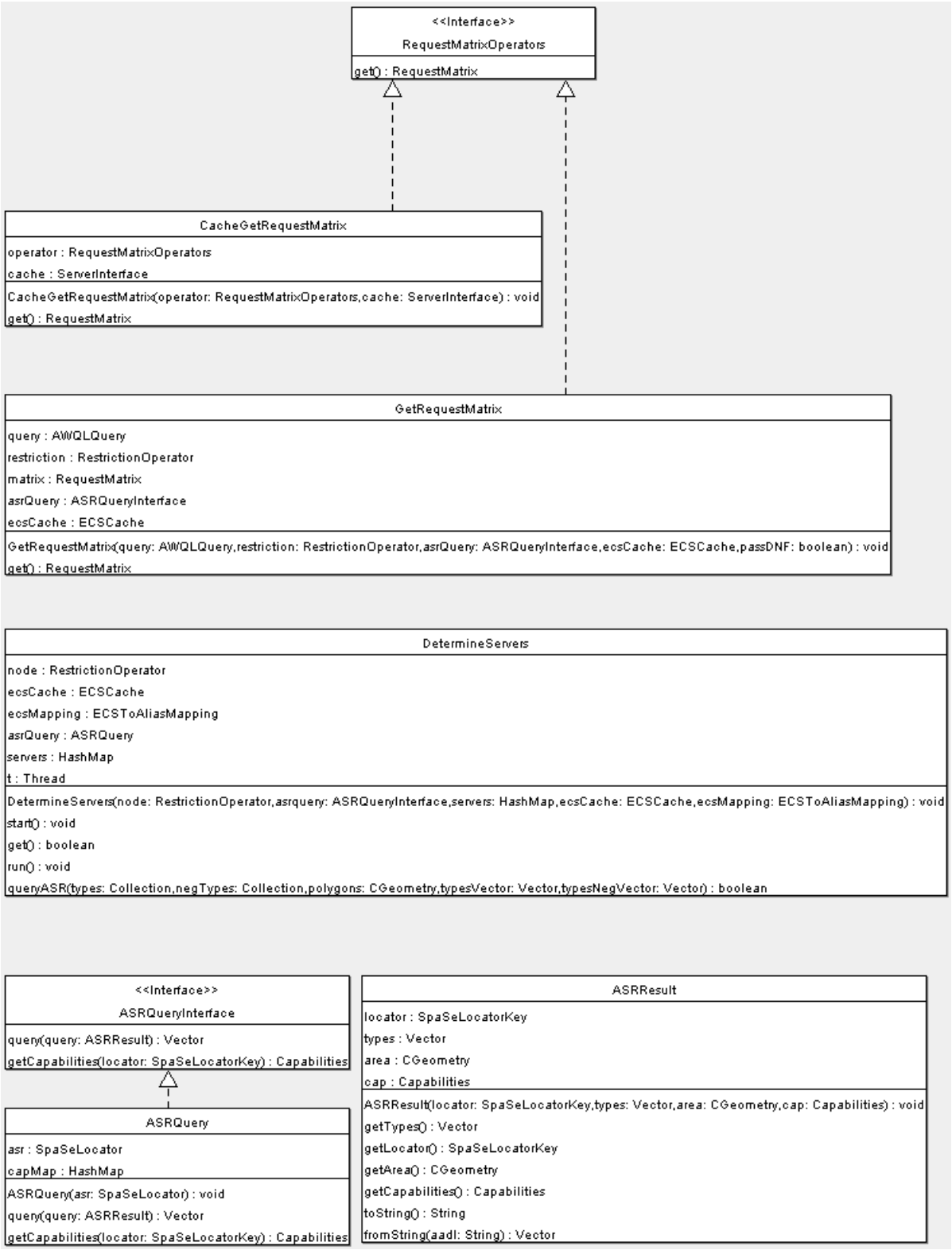


Abb. 48: Klassendiagram der RequestMatrixOperator Klassen

6.3.1 RequestMatrixOperators

get
Diese Methode der RequestMatrix Operatoren holt die *RequestMatrix* des Kindknotens, bearbeitet sie, und gibt sie zurück.

6.3.2 DetermineServers

start
Die Methode startet den Thread

get
Die Methode wartet, bis der Thread beendet ist und gibt ein Flag zurück. Dieses Flag ist *true*, wenn der Operator beim Parsen einen Ausdruck gefunden hat, welcher zur Bestimmung von Servern verwendet werden kann.

run
Der Thread parst eine Konjunktion, um festzustellen, an welche Server diese Konjunktion in einem Request gesendet werden muß. Dazu wird eventuell eine Anfrage an das *ASR* gestellt. Die gefundenen Server werden in *Server* Objekten in einer *HashMap* gespeichert. Der Zugriff auf die *HashMap* ist synchronisiert.

queryASR
Diese Methode stellt eine Anfrage an das *ASR*, entfernt aus der Antwort alle Server welche keine Objekte für den jeweiligen zu parsenden Restriction-Ausdruck beitragen werden und fügt für die verbliebenen Server einen Eintrag der *Servers HashMap* hinzu.

6.3.3 GetRequestMatrix

get
Wurde ein *AAS* Element im *AWQL* Dokument übergeben, dann wird die *RequestMatrix* mithilfe des *AAS* Elements aufgebaut, da die Server damit bereits bekannt sind. Ansonsten startet die Methode für jede Konjunktion einen *ParseConjunction* Operator. Der Thread wartet gegebenenfalls, bis alle *ParseConjunction* Operatoren beendet sind und baut dann aus den *Server* Objekten der *HashMap* das *RequestMatrix* Objekt auf.

6.3.4 CacheGetRequestMatrix

get
Die Methode holt das *RequestMatrix* Objekt vom Kindknoten und übergibt es dem Cache. Der Cache gibt ein *RequestMatrix* Objekt zurück. Dieses *RequestMatrix* Objekt gibt die Methode zurück.

6.3.5 ASRQuery

query
Die Methode führt eine *query* Operation beim *Area Service Register* durch. Der Methode wird das *AADL* Dokument in einem *ASRResult* Objekt übergeben. Die Methode ruft die Methode *toString* dieses Objekts auf, um das *AADL* Dokument zu erhalten. Wurde der Methode nur ein *SpaSeLocatorKey* übergeben und sind alle anderen übergebenen Objekte *null* Referenzen, wenn also die Methode von *getCapabilities* aufgerufen wurde, dann werden mit dem *Result-*

Spec Element nur die Capabilities angefordert. Ansonsten werden zusätzlich auch die Typen der Server angefragt. Die Anfrage wird an das *ASR* übergeben und das vom *ASR* erhaltene *AAList* Dokument wird in einen Vektor mit *ASRResult* Objekten konvertiert, indem die *fromString* Methode dieser Klasse aufgerufen wird. Eventuell gefundene *Capabilities* werden in der *HashMap capMap* gecacht.

getCapabilities
Die Methode gibt das *Capabilities* Objekt eines Servers zurück. Der *SpaSeLocator* des Server wird der Methode übergeben.

Ist das *Capabilities* Objekt noch nicht in der HashMap *capMap* vorhanden, dann wird beim *ASR* nachgefragt. Die Antwort des Cache wird geparst. Das jetzt in der *capMap* gefundene *Capabilities* Objekt wird zurückgegeben. Konnte es erneut nicht gefunden werden, dann wird ein Default *Capabilities* Objekt zurückgegeben (Server sortiert nicht und kann *restriction* Ausdruck parsen).

6.3.6 ASRResult

Ein *ASRResult* Objekt repräsentiert das Ergebnis einer Anfrage beim *Area Service Register* oder eine Anfrage an das *Area Service Register.*

ASRResult
Dem Konstruktor wird der *SpaSeLocator* des Servers, die Nexus-Types, welche dieser zurückgeben kann und die Service Area des Servers übergeben.

getLocator
Die Methode gibt den *SpaSeLocator* des Servers zurück.

getTypes
Die Methode gibt die Types des Servers zurück.

getArea
Die Methode gibt die Service Area des Servers zurück.

toString
Die Methode konvertiert ein *ASRResult* Objekt in ein *AADL* Dokument. Dieses Dokument kann dem *ASR* übergeben werden.

fromString
Die Methode parst die Antwort des *ASR*. Zum Parsen wird der *XML Pull Parser* verwendet. In der Antwort wird für jeden zurückgegebenen Server nach den *Capabilities* und gegebenenfalls nach Klassen gesucht. Die *Capabilities* werden in der *HashMap capMap* eingetragen. Wurden auch Klassen gefunden, dann werden diese zusammen mit dem *SpaSeLocator* des Servers in einem ASRResult Objekt abgelegt. Dieses *ASRResult* Objekt wird in dem Vektor gespeichert, welcher von dieser Methode zurückgegeben wird.

Das *ASR* unterstützt noch nicht, die Capabilities eines Servers abzufragen. Von dieser Klasse wird die Anfrage so implementiert:

Um *Capabilities* Objekte in der Antwort anzufordern wird ein *capabilities* Attribut dem *resultspec* Element hinzugefügt: <resultspec capabilities="yes"/>"

In der Antwort sollen die Capabilities eines Servers jeweils in dem aadl Element so hinzugefügt werden: <capabilities sorted=”false” ignoresRestriction=”false”/>

6.4 Die Query Operatoren

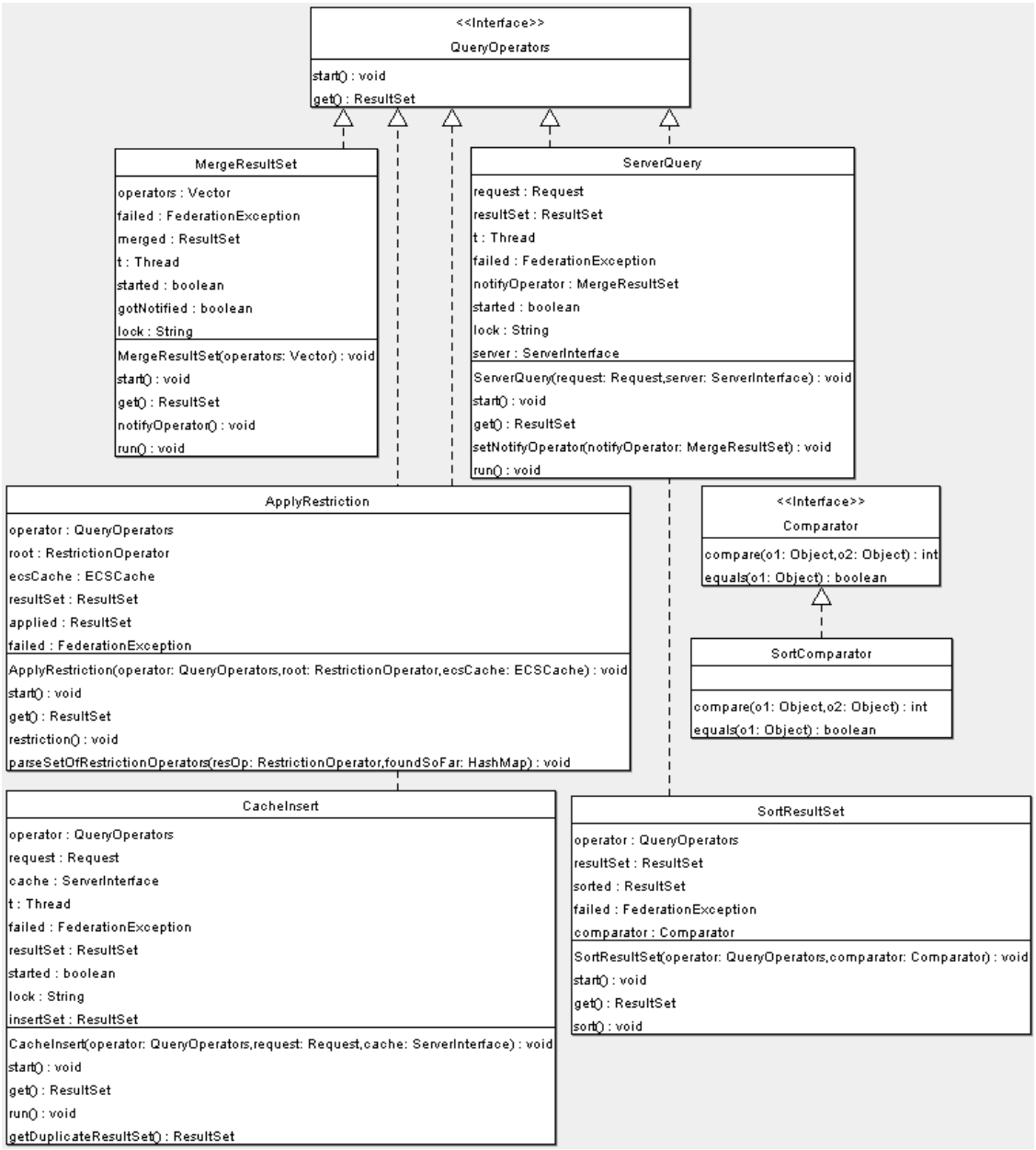


Abb. 49: Klassendiagram der Query Operatoren

6.4.1 QueryOperators

start
Die Methode ruft die *start* Methoden der Kindoperatoren auf und kehrt dann zurück.

get

Die Methode ruft die get Methoden der Kindoperatoren auf, bearbeitet die *ResultSets* und gibt das resultierende *ResultSet* zurück.

6.4.2 MergeResultSet

get
Die Methode wartet darauf, daß der Thread endet und gibt dann das *ResultSet* zurück.

run
Der Thread wartet in einem *synchronized* Block darauf, daß einer der Kind-Operatoren, also ein *ServerQuery* oder *CacheQuery* Operator die *notifyOperator* Methode aufruft. Nachdem der Thread aufgeweckt wurde, überprüft er bei jedem Kind-Operator, ob dieses ein *ResultSet* zurückgibt oder eine *FederationException* sollte ein Fehler aufgetreten sein. Hat der Kind-Operator ein *ResultSet* oder eine *Exception* erhalten, dann wird der Kind-Operator aus einem temporären Vektor entfernt. Wurde ein ResultSet zurückgegeben, dann wird es mit dem gespeicherten vereinigt.

6.4.3 SortResultSet

get
Ist in *failed* eine *FederationException* (des Kind-Operators) gespeichert, dann wird diese zurückgegeben. Die Methode holt das *ResultSet* vom Kind-Operator, sortiert dieses, indem die Methode *sort* aufgerufen wird und gibt das ResultSet zurück. Trat bei dem Aufruf der *get* Methode des Kind-Operators eine *Exception* auf, dann wird diese zusätzlich in *failed* gespeichert.

sort
Die Methode sortiert das *ResultSet* nach der *ObjectId* der Nexus-Objekte.

6.4.4 SortComparator

compare
ist eines der beiden übergebenen Objekte kein *GenericObject* Objekt, dann gibt die Methode eine *ClassCastException* Exception zurück. Hat *o1* eine größere *ObjectId* als *o2*, dann wird ein positiver integer Wert zurückgegeben. Haben beide die selbe *ObjectId*, dann wird 0 zurückgegeben. Hat *o1* eine kleinere *ObjectId* als *o2*, dann wird ein negativer integer Wert zurückgegeben.

equals
Ist o1 ein SortComparator Objekt, dann wird true zurückgegeben.

6.4.5 CacheInsert

get
Ist *failed* nicht null, dann wird die *FederationException* zurückgegeben. Die Methode ruft die *get* Methode des Kind-Operators auf. Hat der Kind-Operator eine *Exception* zurückgegeben, dann wird diese in *failed* abgespeichert. Der Thread wird in diesem Fall danach aufgeweckt. Wurde dagegen ein *ResultSet* zurückgegeben, dann wird eine Kopie davon in *insertSet* erstellt und der Thread aufgeweckt. Das *ResultSet* wird zurückgegeben.

run

Der Thread wartet, bis die Methode *notifyOperator* aufgerufen wird. Danach wird überprüft, ob *insertSet* gesetzt ist. Ist das der Fall, dann wird die Methode *insert* des *Cache* aufgerufen.

getDuplicateResultSet
Diese Methode wird von der *get* Methode Operators verwendet. Sie gibt ein Duplikat des *ResultSets* zurück. Dieses Duplikat des *ResultSet* wird dem *Cache* übergeben, damit der *Cache* und die *Föderation* das *ResultSet* gleichzeitig und unabhängig voneinander verwenden und verändern können.

6.4.6 ApplyRestriction

get
Ist in *failed* eine *FederationException* (des Kind-Operators) gespeichert, dann wird diese zurückgegeben. Die Methode ruft die get Methode des Kind-Operators auf. Wird ein *ResultSet* zurückgegeben, dann wird die Methode *restriction* aufgerufen. Das Objekt *applied* wird zurückgegeben. Wurde von get *null* zurückgegeben und daher die Methode *restriction* nicht aufgerufen, dann wird daher *null* zurückgegeben.

restriction
Die Methode iteriert über alle Konjunktionen des *RestrictionOperator* Objekts und ruft zum Parsen der einzelnen Konjunktion (oder eines Elements) die Methode *parseSetOfRestrictionOperators* auf. Die Objekte in *foundSoFar* werden in einen neues *ResultSet* eingetragen und dieses *applied* zugewiesen.

parseSetOfRestrictionOperators
Die Methode parst eine Konjunktion. Der Methode wird der zu parsende RestrictionOperator und eine *HashMap foundSoFar* übergeben. In der HashMap werden die *GenericObject* Objekte gesspeichert, welche eine der Konjunktionen erfüllen. Das ResultSet *resultSet* wird einem ResultSet *foundResultSet* zugewiesen. Die Methode parst zuerst alle nicht invertierten Elemente dieser Konjunktion. Kann zu einem ResultSetFilter Objekt keine weiteren Bedingungen hinzugefügt werden, dann wird die *query* Methode des *ResultSets foundResultSet* aufgerufen und das Ergebnis wieder im *ResultSet foundResultSet* abgespeichert. Danach werden nur die invertierten Elemente geparst. Nach jedem Element wird ein *ResultSetFilter* der Methode *queryIterator* des *ResultSets foundResultSet* übergeben. Jedes gefundene *GenericObject* wird aus dem *ResultSet foundResultSet* entfernt. Nachdem alle Elemente geparst sind werden alle Objekte in *foundResultSet* in *foundSoFar* eingetragen aus *resultSet* entfernt, da diese Objekte nicht mehr in anderen Konjunktionen überprüft werden müssen.

6.4.7 ServerQuery

start
Die Methode startet den Thread.

get
Ist failed *true*, dann wird eine *FederationException* zurückgegeben. Läuft der Thread noch, dann wird eine *null* Referenz zurückgegeben. Ansonsten wird das *ResultSet* zurückgegeben.

run
Der Thread übergibt das *Request* Objekt dem Objekt mit der *ServerInterface* Schnittstelle, welches dem Konstruktor dieses Objekts übergeben wurde. Dies kann ein Objekt, welches per *ApacheSOAP* mit einem Server kommuniziert oder der Cache sein. Das von diesem Objekt

erhaltene *ResultSet* wird abgespeichert.

Tritt bei der Kommunikation mit dem Server ein Fehler auf, dann wird *failed* auf *true* gesetzt.

6.5 Die QueryOperatorsCRL Operatoren

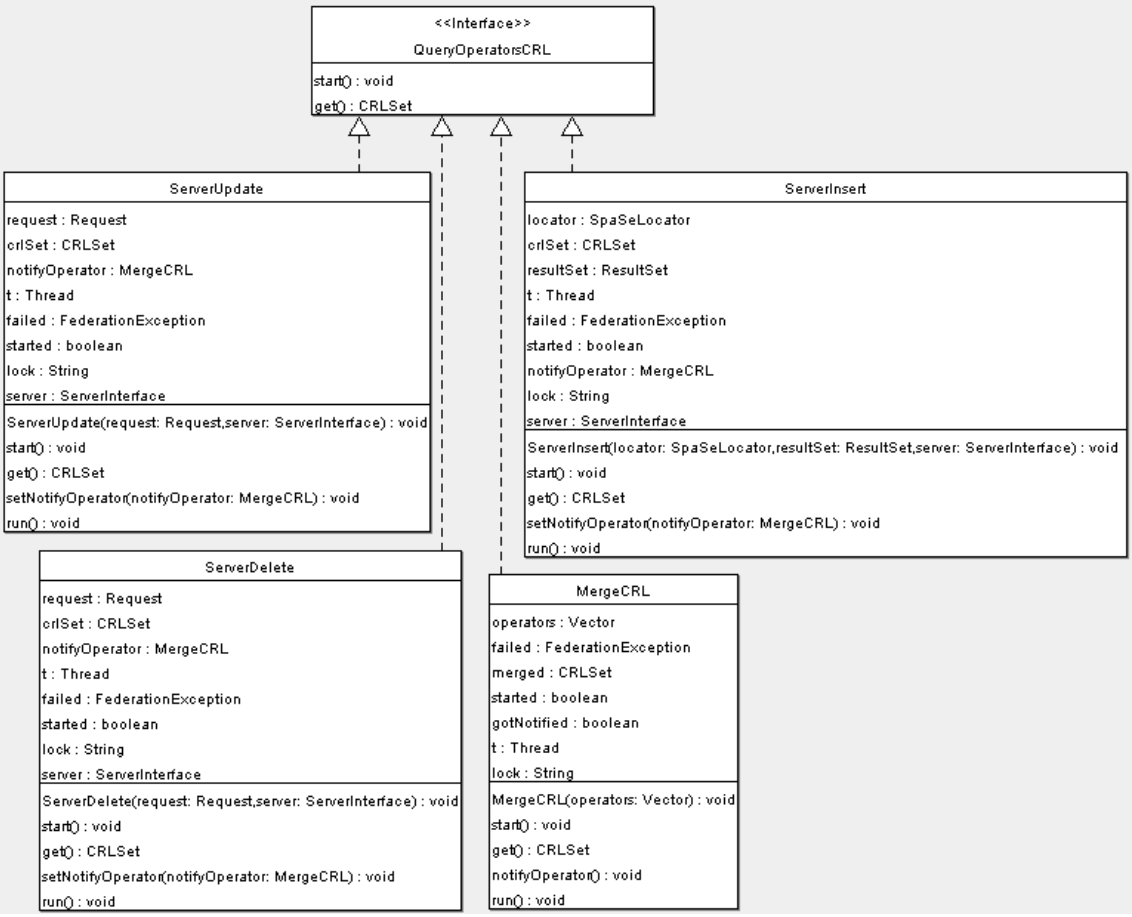


Abb. 50: Klassendiagram der QueryCRL Operatoren

6.5.1 QueryOperatorsCRLOperators

start
Die Methode ruft die *start* Methoden der Kindoperatoren auf und kehrt dann zurück.

get
Die Methode ruft die *get* Methoden der Kindoperatoren bzw. des Kindoperators auf, bearbeitet alle *CRLSetWrapper* Objekte und gibt das resultierende *CRLSetWrapper* Objekt zurück.

6.5.2 MergeCRL

Analog zu *MergeResultSet* vereint dieser Operator *CRLSetWrapper* Objekte seiner Kindoperatoren.

6.5.3 ServerUpdate

Analog zu dem *ServerQuery* Operator übergibt dieser Operator eine Anfrage an eine Server.

Dieser Operator allerdings übergibt die Anfrage an die *update* Methode und gibt ein *CRLSet* Objekt zurück.

6.5.4 ServerInsert

So wie auch der ServerQuery Operator übergibt dieser Operator eine Anfrage an einen Server. Dieser Operator übergibt die einzufügenden Nexus Objekte dem Server und gibt ein *CRLSet* Objekt zurück.

6.5.5 ServerDelete

Analog zu dem *ServerQuery* Operator übergibt dieser Operator eine Anfrage an eine Server. Dieser Operator allerdings übergibt die Anfrage an die *delete* Methode und gibt ein *CRLSet* Objekt zurück.

6.6 CRL Klassen

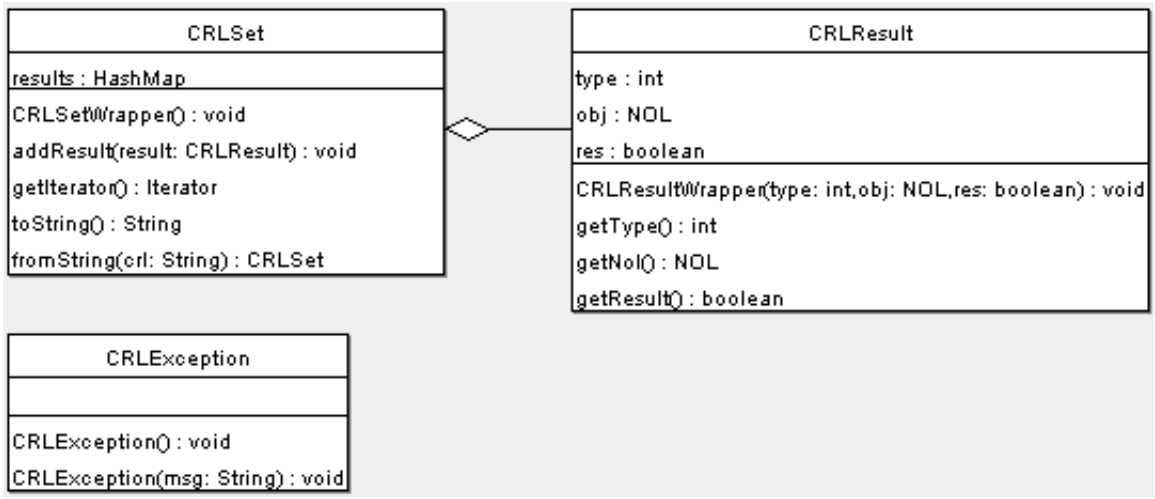


Abb. 51: Klassendiagram der *CRL* Klassen

Die *CRL* Klassen werden dazu verwendet *CRL* Dokumente und dessen Einträge zu repräsentieren. Die Klasse *CRLSet* kann *CRL* Dokumente parsen und erzeugen.

6.6.1 CRLSet

Die Klasse *CRLSet* repräsentiert jeweils ein *CRL* Dokument. Die einzelnen Einträge des *CRL* Dokuments, also die *action* Elemente, werden mit *CRLResult* Objekten repräsentiert.

addResult
Die Methode fügt dem *CRLSet* Objekt ein weiteres *CRLResult* Objekt hinzu.

getIterator
Die Methode gibt ein *Iterator* Objekt über die im *CRLSet* gespeicherten *CRLResult* Objekte zurück.

toString
Die Methode gibt das durch das *CRLSet* Objekt repräsentierte *CRL* Dokument in einem *String*

Objekt zurück.

fromString
Die statische Methode erzeugt aus einem übergebenen *CRL* Dokument ein *CRLSet* Objekte. Tritt beim Parsen ein Fehler auf, dann gibt die Methode eine *CRLException* zurück.

6.6.2 CRLResult

CRLResult
Der Konstruktor.

getType
Die Methode gibt den Typ des *action* Elements zurück. Dies kann *CRLResult.UPDATE*, *CRL-Result.INSERT* oder *CRLResult.DELETE* sein.

getNol
Die Methode gibt die *NOL* des Nexus Objekts zurück, für welches das *action* Element angibt. ob eine Anfrage erfolgreich war oder nicht.

getResult
Die Methode gibt zurück, ob die Anfrage erfolgreich war oder nicht.

6.6.3 CRLException

Tritt beim Parsen eines *CRL* Dokuments oder beim Instaziieren eines *CRLResult* Objekts ein Fehler auf, dann wird diese Exception zurückgegeben.

6.7 Sonstige Klassen

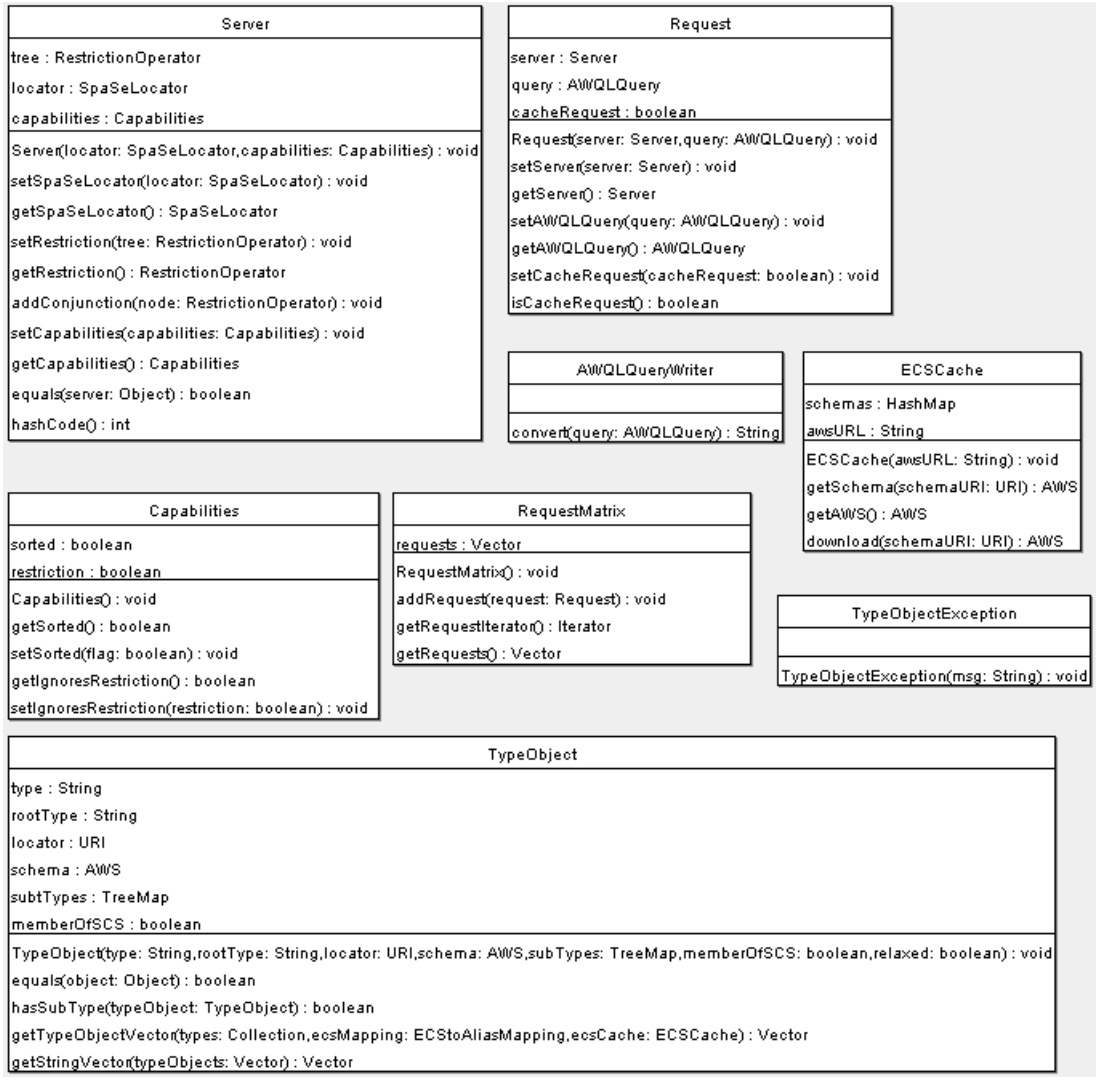


Abb. 52: Klassendiagram der Sonstigen Klassen

6.7.1 Server

Ein Objekt der Klasse *Server* repräsentiert einen Server, dessen Capabilities und die Konjunktionen, welche in einem Request an ihn zu senden sind.

Server
Dem Konstruktor muß der *SpaSeLocator* und die Capabilities übergeben werden.

setSpaSeLocator
Die Methode ändert den *SpaSeLocator* des Objekts.

getSpaSeLocator
Die Methode gibt den *SpaSeLocator* des Objekts zurück.

setRestriction
Die Methode ersetzt den bisherigen Restriction Ausdruck des Objekts. Dies kann eine *DNF*

sein.

getRestriction
Die Methode gibt den Restriction Ausdruck des Objekts zurück.

addConjunction
Die Methode fügt dem Restriction Ausdruck des Objekts eine Konjunktion oder ein Element hinzu. Ist der bisherige Restriction Ausdruck noch nicht vorhanden, also wenn zu Beispiel diese Methode das erste mal aufgerufen wird, dann wird das übergebene Objekt der Restriction Ausdruck des Objekts. Ist dagegen ein Restriction Ausdruck bereits vorhanden und keine *DNF*, dann wird einer *DNF* erzeugt und der bisherige Restriction Ausdruck und die übergebene Konjunktion dieser *DNF* hinzugefügt. Ist der Restriction Ausdruck bereits eine *DNF*, dann wird die Konjunktion dieser *DNF* hinzugefügt.

setCapabilities
Die Methode setzt die Capabilities Flags des Objekts.

getCapabilities
Die Methode gibt die Capabilities Flags des Objekts zurück.

equals
Die Methode wird implementiert, um ein Objekt dieser Klasse in einer HashMap abspeichern zu können. Die Objekte gelten dann als gleich, wenn die *SpaSeLocator* Objekte gleich sind.

hashCode
Die Methode wird implementiert, um ein Objekt dieser Klasse in einer HashMap abspeichern zu können. Der Hash-Code wird aus dem *SpaSeLocator* berechnet.

6.7.2 Request

Ein Objekt der Klasse *Request* repräsentiert eine *AWQL* Anfrage. Die Objekte dieser Klasse werden in *RequestMatrix* Objekte verwendet.

Request
Dem Konstruktor müssen der *SpaSeLocator* des Servers, die Capabilities des Servers und das *AWQL* Dokument übergeben werden.

setServer
Die Methode ändert das *Server* Objekt des Objekts.

getServer
Die Methode gibt das *Server* Objekt des Objekts zurück.

setAWQLQuery
Die Methode ändert das *AWQLQuery* Objekt Objekts.

getAWQLQuery
Die Methode gibt *AWQLQuery* Objekt des Objekts zurück.

setCacheRequest
Die Methode setzt das Flag, welches diesen Request als eine Request markiert, welcher an den Cache der Föderation und nicht an einen Nexus Server der Dienste Schicht gestellt werden

soll.

getCacheRequest
Die Methode list das durch *setCacheRequest* gesetzte Flag.

6.7.3 RequestMatrix

Ein Objekt dieser Klasse repräsentiert eine Menge von *AWQL* Requests. Objekte dieser Klasse werden zur Anfrage beim Cache verwendet und werden dabei vom Cache zurückgegeben.

addRequest
Die Methode fügt dem *RequestMatrix* Objekt ein weiteres *Request* Objekt hinzu.

getRequestIterator
Die Methode gibt ein *Iterator* Objekt über alle *Request* Objekt zurück.

getRequests
Die Methode gibt alle *Request* Objekte in einem *Vektor* zurück.

6.7.4 Capabilities

Ein Objekt dieser Klasse repräsentiert bestimmte Eigenschaften eines Servers. Mit einem Objekt dieser Klasse kann zum Beispiel angegeben werden, ob ein Server die Nexus Objekte im AWML Dokument bereits nach der ObjectId sortiert zurückgibt oder ob der Restriction Ausdruck in dem AWQL Dokument, welches dem Server übergeben wird, auch berücksichtigt wird. Weitere Flags können der Klasse bei Bedarf hinzugefügt werden.

setSorted
Mit dieser Methode wird das Flag gesetzt, welches angibt, ob der Server die Nexus Objekte nach der ObjectId sortiert zurückgibt.

getSorted
Mit dieser Methode wird das Flag ausgelesen.

setIgnoresRestriction
Mit dieser Methode wird das Flag gesetzt, welches angibt, ob der Server den Restriction Ausdruck des an ihn übergebenen AWQL Ausdrucks ignoriert.

getIgnoresRestriction
Mit dieser Methode wird das Flag ausgelesen.

6.7.5 ECSCache

getSchema
Die Methode gibt das Schema als *AWS* Objekt, welches unter der übergebenen *URI* (eine *URL*) gefunden werden kann zurück. Dazu wird die Methode *download* aufgerufen, falls das *AWS* Objekt noch nicht im Cache ist.

getAWS
Die Methode gibt das Schema *'http://nexus.uni-stuttgart.de/0.1/AugmentedWorldSchema'* zurück. Dazu wird die Methode *download* aufgerufen, falls das *AWS* Objekt noch nicht im Cache ist.

download
Die Methode downloadet das Schema, konvertiert es in ein *AWS* Objekt und speichert es im Cache ab. *HTTP-redirects* werden bis zu fünf mal gefolgt. Das AWS Objekt wird, auch bei einem *HTTP-redirect*, unter der ursprünglichen *URI* im Cache gespeichert.

6.7.6 ContactServer

Dieses Objekt implementiert das *ServerInterface* Interface. Die Methoden übergeben die Requests an die entsprechenden Methoden eines Nexus Servers per *ApacheSOAP.*

queryRequestMatrix
Diese Methode enthält keinen Code, da diese Methode nur für den *Cache* aufgerufen wird.

query
Die Methode liest das *AWQLQuery* Objekt aus. Der *SpaSeLocator* und das *ResultSet* werden aus dem *Server* Objekt ausgelesen. Um das *AWQL* Dokument aufzubauen, wird dieses *ResultSet* und nicht das *ResultSet* des *AWQLQuery* Objekts verwendet. Das *AWQL* Dokument wird dem Server per *ApacheSOAP* übergeben und das erhaltene *AWML* Dokument wird in ein *ResultSet* konvertiert und zurückgegeben.

update
Die Methode liest das *AWQLQuery* Objekt aus. Der *SpaSeLocator* und das *ResultSet* werden aus dem *Server* Objekt ausgelesen. Um das *AWQL* Dokument aufzubauen, wird dieses *ResultSet* und nicht das *ResultSet* des *AWQLQuery* Objekts verwendet. Das *AWQL* Dokument wird dem Server per *ApacheSOAP* übergeben und das erhaltene *CRL* Dokument wird in ein *CRLSet* konvertiert und zurückgegeben.

insert
Die Methode liest den *SpaSeLocator* aus dem *Server* Objekt des *Request* Objekts aus. Das *ResultSet* wird in ein *AWML* Dokument konvertiert. Das *AWML* Dokument wird dem Server per *ApacheSOAP* übergeben und das erhaltene *CRL* Dokument wird in ein *CRLSet* konvertiert und zurückgegeben.

delete
Die Methode liest das *AWQLQuery* Objekt aus. Der *SpaSeLocator* und das *ResultSet* werden aus dem *Server* Objekt ausgelesen. Um das *AWQL* Dokument aufzubauen, wird dieses *ResultSet* und nicht das *ResultSet* des *AWQLQuery* Objekts verwendet. Das *AWQL* Dokument wird dem Server per *ApacheSOAP* übergeben und das erhaltene *CRL* Dokument wird in ein *CRLSet* konvertiert und zurückgegeben.

6.7.7 TypeObject

Ein Objekt dieser Klasse repräsentiert einen *type*, wie er in einem *equal* Element eines *AWQL* Dokuments vorkommt. In dem Objekt wird der *type* und ein Alias falls vorhanden abgespeichert. Ebenfalls werden alle Sub-Typen des Typs (im selben Schema), der Wurzel-Typ der Typhierarchie, das Schema, die *URI* des Schemas und ein Flag welches angibt, ob der Typ ein Typ im SCS ist gespeichert.

equals
Gibt true zurück, wenn beide Typen identisch sind.

hasSubType
Gibt true zurück, wenn der übergebene Typ ein Sub-Typ des Typs ist.

getTypeObjectVector
Konvertiert ein Vector mit Typen in einen Vektor mit TypeObjects. Hat ein *type* keinen Alias, ist in dem *AWQL* Dokument ein *scope* Element vorhanden und wird der type in mehreren ECS gefunden, dann werden für alle in den Schemata gefundenen typen Einträge dem Vektor hinzugefügt.

getStringVector
Die Methode konvertiert einen Vektor mit TypeObjects in einen Vektor mit String Objekten.

6.7.8 TypeObjectException

Dieses Objekt wird von der Klasse TypeObject zurückgegeben, wenn ein Fehler aufgetreten ist.

7 Komponententests

In diesem Kapitel sollen für einige Komponenten der Föderation die Performance gemessen werden und die Funktionalität überprüft werden. Dazu werden die Klassen *ReadFiles* und *ASRFile* implementiert. Die Klasse *ReadFiles* implementiert *ServerInterface* und die Klasse *ASRFile* implementiert *ASRQueryInterface*. Die Klassen lesen die Dokumente aus Dateien aus. Dazu verwenden sie die Klasse *TestHelperClass*.

Um die *DNFOperators* Operatoren zu testen, wird ein *DisplayRestriction DNFOperators* Operator implementiert, welcher die Restriction mit der Klasse *LogWriter* nach *stdout* ausgibt. Das Schema der *AWS* wird von *http://127.0.0.1:8080/aws-0.1.xsd* heruntergeladen. Das Schema wird mit *Apache Tomcat* gehostet.

7.1 Die Hilfsklassen.

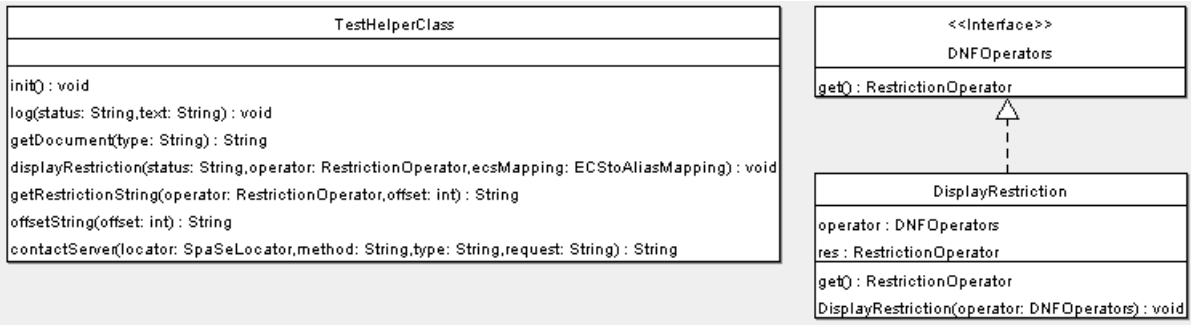


Abb. 53: Klassendiagram der Hilfsklassen

7.1.1 TestHelperClass

init
Die Methode *getDocument* kann ein *AWQL* Dokument, mehrere *AWML* Dokumente, mehrere CRL Dokumente und ein *AALIST* Dokument zurückgeben. Bei jeder Rückgabe eines *AWML* oder CRL Dokuments wird ein interner Zähler inkrementiert, welcher dafür sorgt, daß die *AWML* Dateien und CRL Dateien aus *awml<counter1>.txt* bzw. *crl<counter2>.txt* ausgelesen werden. Diese Methode setzt die Zähler auf 0 zurück.

log
Die Methode gibt den Statustext und den Text nach *stdout* aus. Da diese Methode wie auch alle anderen Methoden dieser Klasse *synchronized* (und *static*) ist, erfolgt die Ausgabe nach *stdout* atomar.

getDocument
Dieser Methode wird der String "*awql*", "*awml*", "*crl*" oder "*asr*" übergeben. Die Methode liest das entsprechende Dokument aus *awql.txt*, *awml<counter1>.txt*, *crl<counter2>.txt* oder *asr.txt* aus und gibt es in einem *String* Objekt zurück.

displayRestriction
Diese Methode gibt den übergebenen *RestrictionOperator*-Baum als Text nach *stdout* aus. Dazu wird die *log* Methode dieser Klasse verwendet.

getRestrictionString
Diese Methode wird von der Methode *displayRestriction* verwendet. *getRestrictionString* ruft sich selbst rekursiv auf.

offsetString
Die Methode wird von *getRestrictionString* verwendet. Sie gibt einen *String* mit der übergebenen Anzahl von Leerzeichen zurück.

contactServer
Die Methode ruft per *ApacheSOAP RPC* die Methode *method* bei dem durch *locator* angegebenen Objekt/Server auf und übergibt der Methode das *request String*-Objekt. Die Antwort in einem *String* Objekt oder eine *SOAPException* wird zurückgegeben.

7.1.2 DisplayRestriction

get
Die Methode holt den *RestrictionOperator* des Kindoperators, gibt diesen unter Verwendung der Klasse *TestHelperClass* nach *stdout* aus und gibt den *RestrictionOperator* zurück.

DisplayRestriction
Der Konstruktor

7.2 Die Tests

Um die Funktionalität der Komponenten festzustellen, wurde eine Testklasse implementiert, welche die Methoden der Klasse Federation direkt aufruft. Die Dokumente werden alle mit der *TestHelperClass* aus Dateien geladen. Die Tests wurden unter Windows 2000 SP4 und Java2 SDK 1.4.2_03 auf einem AMD Athlon 900Mhz durchgeführt.

Um die *DNFOperatoren*, die *RequestMatrix* Operatoren, die *QueryOperator* Operatoren und die sonstigen Klassen zu testen, wurden einige *AWQL* Dokumente der Föderation übergeben (die untenstehenden Dokumente verwenden Anfragen aus diesen Dokumenten), die jeweils eine bestimmte Funktionalität bestimmter Operatoren testeten. Dabei zeigte sich, daß die Föderation diese Anfragen im Rahmen des Tests korrekt verarbeitet.

7.2.1 Performancetests der DNFOperators Operatoren

Dem *DNFOperators* Baum wurden mehrere verschieden komplexe *AWQL* Dokumente übergeben. Der Baum wird in einer Schleife 1.000.000 mal mit einer Kopie des *RestrictionOperator* Baums aufgerufen um die durchschnittliche Bearbeitungsdauer zu bestimmen:

Dokument1:

```
<awql>
 <restriction>
  <or>
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000001</NOL></nexusdata>
    </equal>
```

```
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.2:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000002</NOL></nexusdata>
    </equal>
  </or>
 </restriction>
</awql>
```

Dokument 2:

```
<awql>
 <restriction>
  <or>
   <and>
    <in>
     <attr name="type"/>
     <nexusdata><String>room</String></nexusdata>
     <nexusdata><String>mobileobject</String></nexusdata>
    </in>
    <not>
     <not>
      <or>
       <equal>
        <attr name="nol"/>
        <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                       urn:QueryComponent|0x00000000000000000000000000000001/
                        0x00000000000000000000000000000001</NOL></nexusdata>
        </equal>
        <equal>
         <attr name="nol"/>
         <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                        urn:QueryComponent|0x00000000000000000000000000000001/
                         0x00000000000000000000000000000002</NOL></nexusdata>
        </equal>
      </or>
     </not>
    </not>
   </and>
   <or>
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000003</NOL></nexusdata>
    </equal>
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000004</NOL></nexusdata>
    </equal>
   </or>
  </or>
 </restriction>
</awql>
```

Dokument 3:

```
<awql>
 <restriction>
  <and>
```

```
   <or>
    <in>
     <attr name="type"/>
     <nexusdata><String>room</String></nexusdata>
     <nexusdata><String>mobileobject</String></nexusdata>
    </in>
    <not>
     <not>
      <or>
       <equal>
        <attr name="nol"/>
         <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                         urn:QueryComponent|0x00000000000000000000000000000001/
                          0x00000000000000000000000000000001</NOL></nexusdata>
        </equal>
        <equal>
         <attr name="nol"/>
         <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                         urn:QueryComponent|0x00000000000000000000000000000001/
                          0x00000000000000000000000000000002</NOL></nexusdata>
        </equal>
      </or>
     </not>
    </not>
   </or>
   <or>
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                           urn:QueryComponent|0x00000000000000000000000000000001/
                            0x00000000000000000000000000000003</NOL></nexusdata>
    </equal>
    <equal>
     <attr name="nol"/>
     <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000004</NOL></nexusdata>
    </equal>
   </or>
  </and>
 </restriction>
</awql>
```

Die Dokumente wurden in diese Dokumente konvertiert:

Ergebnis 1:

```
<in>
  <attr name="nol"/>
  <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                  urn:QueryComponent|0x00000000000000000000000000000001/
                   0x00000000000000000000000000000001</NOL></nexusdata>
  <nexusdata><NOL>nexus:http://127.0.0.2:8081/soap/servlet/rpcrouter|
                  urn:QueryComponent|0x00000000000000000000000000000001/
                  0x00000000000000000000000000000002</NOL></nexusdata>
</in>
```

Ergebnis 2:

```
<or>
  <in>
    <attr name="nol"/>
    <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
```

```
                    urn:QueryComponent|0x00000000000000000000000000000001/
                    0x00000000000000000000000000000003</NOL></nexusdata>
    <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                    urn:QueryComponent|0x00000000000000000000000000000001/
                    0x00000000000000000000000000000004</NOL></nexusdata>
  </in>
  <and>
    <in>
      <attr name="type"/>
      <nexusdata><String>room</String></nexusdata>
      <nexusdata><String>mobileobject</String></nexusdata>
    </in>
    <in>
      <attr name="nol"/>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000001</NOL></nexusdata>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000002</NOL></nexusdata>
    </in>
  </and>
</or>
```

Ergebnis 3:

```
<or>
  <and>
    <equal>
      <attr name="nol"/>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000004</NOL></nexusdata>
    </equal>
    <in>
      <attr name="nol"/>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000001</NOL></nexusdata>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000002</NOL></nexusdata>
    </in>
  </and>
  <and>
    <equal>
      <attr name="nol"/>

      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000003</NOL></nexusdata>
    </equal>
    <in>
      <attr name="nol"/>

      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000001</NOL></nexusdata>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                      urn:QueryComponent|0x00000000000000000000000000000001/
                      0x00000000000000000000000000000002</NOL></nexusdata>
    </in>
  </and>
  <and>
```

```
    <in>
      <attr name="type"/>
      <nexusdata><String>room</String></nexusdata>
      <nexusdata><String>mobileobject</String></nexusdata>
    </in>
    <in>
      <attr name="nol"/>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                     urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000003</NOL></nexusdata>
      <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                     urn:QueryComponent|0x00000000000000000000000000000001/
                     0x00000000000000000000000000000004</NOL></nexusdata>
    </in>
  </and>
</or>
```

Für die drei Dokumente wurden folgende Laufzeiten gemessen:

Table 3: Laufzeiten des Performancetests der DNFOperator Operatoren

	Laufzeit für 1.000.000 Iterationen	Durchschnittliche Laufzeit
Dokument 1	38646 ms.	0,039 ms.
Dokument 2	119202 ms.	0,119 ms.
Dokument 3	324096 ms.	0,324 ms.

Im Dokument 3 wurde eine *oder* in eine *and* und eine *and* in eine *oder* Verknüpfung geändert. Dadurch erhöhte sich der Aufwand den Ausdruck in eine DNF zu konvertieren, wodurch die Laufzeit stieg. Die Laufzeit der *DNF* Operatoren ist trotzdem immer noch sehr gering, hängt allerdings von dem Aufbau des *Restriction* Ausdrucks ab.

7.2.2 Performancetest der RequestMatrix Operatoren

Den *RequestMatrix* Operatoren werden eine Menge von Konjunktionen unterschiedlichen Typs übergeben. Der Baum wird in einer Schleife 10.000 mal aufgebaut und ausgeführt um die durchschnittliche Bearbeitungsdauer zu bestimmen:

Konjunktionen 1-3:

```
<equal>
 <attr name="nol"/>
 <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                 urn:QueryComponent|0x00000000000000000000000000000001/
                  0x00000000000000000000000000000001</NOL></nexusdata>
</equal>
<and>
 <equal>
  <attr name="type"/>
  <nexusdata><String>room</String></nexusdata>
 </equal>
 <equal>
  <attr name="nol"/>
  <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                 urn:QueryComponent|0x00000000000000000000000000000001/
                  0x00000000000000000000000000000002</NOL></nexusdata>
 </equal>
```

```
</and>
<and>
 <equal>
  <attr name="type"/>
  <nexusdata><String>buildingelement</String></nexusdata>
 </equal>
 <equal>
  <attr name="nol"/>
  <nexusdata><NOL>nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|
                  urn:QueryComponent|0x00000000000000000000000000000001/
                   0x00000000000000000000000000000003</NOL></nexusdata>
 </equal>
</and>
```

Konjunktion 4:

```
<in>
 <attr name="type"/>
 <nexusdata><String>staticobject</String></nexusdata>
 <nexusdata><String>mobileobject</String></nexusdata>
</in>
```

Konjunktion 5:

```
<and>
 <in>
  <attr name="type"/>
  <nexusdata><String>room</String></nexusdata>
  <nexusdata><String>mobileobject</String></nexusdata>
 </in>
 <not>
  <equal>
   <attr name="type"/>
   <nexusdata><String>buildingelement</String></nexusdata>
  </equal>
 </not>
</and>
```

Für die Konjunktionen wurden folgende Laufzeiten gemessen:

Table 4: Laufzeiten des Performancetests der RequestMatrix Operatoren

Konjunktion	Laufzeit 10.000 Iterationen	Durchschnittliche Laufzeit
1-3	22863 ms.	2,3 ms.
1-5	144538 ms.	14,5 ms.
4-5	125270 ms.	12,5 ms.
4	60908 ms.	6,1 ms.
5	65224 ms.	6,5 ms.

Die Laufzeit der *RequestMatrix* Operatoren ist deutlich höher als die Laufzeit der *DNF* Operatoren. Die Konjunktionen 1-3 benötigen nur 2,3 ms., die Konjunktionen 4 und 5 allerdings 6,1 ms. bzw. 6,5 ms. Beim Parsen dieser Konjunktionen muß die Föderation Anfragen an das *Area Service Register* stellen, bzw. in diesem Performance-Test eine Antwort aus einer Datei einlesen. Da keine negierten Typen in einer Anfrage an das *Area Service Register* übergeben werden können, müssen Server aus der Antwort das *Area Service Register* entfernt werden, welche

der Konjunktion keine Objekte liefern werden. Dies kostet zusätzlich Rechenzeit.

7.2.3 Performancetest der QueryOperators Operatoren

Um die Performance der *QueryOperators* Operatoren zu messen, werden Anfragebäume mit einer unterschiedlichen Anzahl von Datenquellen, also *ServerQuery* Operatoren, aufgebaut. Die *ServerQuery* Operatoren geben je nach Test eine unterschiedliche Anzahl von Nexus Objekten zurück. Alle Tests werden mit und ohne *ApplyRestriction* Operatoren und ohne *SortResultSet* Operator durchgeführt. Die eingelesenen *AWML* Dokumente sind so aufgebaut, daß der *ApplyRestriction* Operator 10 Prozent der Objekte weitergibt. Ein Anfragebaum wird in einer Schleife 100 mal aufgebaut und ausgeführt um die durchschnittliche Bearbeitungsdauer zu bestimmen:

Table 5: Laufzeiten des Performancetests der QueryOperators Operatoren

Test	ApplyRestriction	Anzahl der Server	Nexus Objekte (AWML)	Laufzeit 100 Iterationen	Durchschnittliche Laufzeit
1	Nein	15	100	557531 ms.	5575 ms.
2	Nein	10	100	331006 ms.	3310 ms.
3	Nein	8	100	253054 ms.	2530 ms.
4	Nein	5	100	137858 ms.	1379 ms.
5	Ja	5	100	97190 ms.	972 ms.
6	Nein	2	100	49191 ms.	492 ms.
7	Ja	2	100	41961 ms.	420 ms.
8	Nein	5	10	10695 ms.	107 ms.
9	Ja	5	10	8803 ms.	88 ms.
10	Nein	2	10	4537 ms.	45 ms.
11	Ja	2	10	4076 ms.	41 ms.
12	Nein	1	500	95678 ms.	957 ms.
13	Ja	1	500	105192 ms.	1052 ms.
14	Nein	1	1	1523 ms.	15 ms.
15	Ja	1	1	1623 ms.	16 ms.

7.3 Bewertung der Performancetests

Die Kosten der *DNF* Operatoren ist im Vergleich zu den *RequestMatrix* Operatoren und den *Query* Operatoren gering. Die Zeitdauer hängt stark davon ab, wie sehr der Ausdruck verändert werden muß.

Die Kosten der *RequestMatrix* Operatoren sind im Vergleich zu den *Query* Operatoren auch relativ klein, wenn von den *Query* Operatoren viele Objekte von vielen Servern zurückgege-

ben werden. Bei den *RequestMatrix* Operatoren ist besonders das Parsen von Konjunktionen mit *type* Attributen (und Polygonen) zeitaufwendig, da bei diesen Konjunktionen die *queryASR* Methode aufgerufen wird. Diese Methode stellt eine Anfrage an das ASR und entfernt die Server aus der Antwort, welche für diese Konjunktion keine Nexus Objekte liefern können. Dabei wird die Klassenhierarchie des *AWS* verwendet. Da bei der Konjunktion 5 im Gegensatz zur Konjunktion 4 kein Entfernen von Servern notwendig ist, wurde diese geringfügig schneller vom *ParseConjunction* Operator abgearbeitet. Bei einem größeren *AWS* wird der Unterschied noch deutlicher sein.

Die *Query* Operatoren verbrauchen am meisten Rechenzeit im Vergleich zu den anderen Operatoren besonders dann, wenn viele Nexus Objekte von vielen Servern zurückgegeben werden. Bei jedem erneuten Mischen von *ResultSets* mit *ResultSetMerger* in *MergeResultSet* steigt die Verarbeitungszeit des Mischens mit der Anzahl der Generic Objects im bereits bestehenden *ResultSet*. Der *ResultSetMerger* profitiert in der jetzigen Implementierung (noch) nicht von nach der *ObjectId* sortierten *ResultSets*. Wird die Implementierung des *ResultSets* dementsprechend geändert, dann kann der *SortResultSet* Operator dazu verwendet werden, die *ResultSets* vor dem Vereinigen zu sortieren.

Um 500 Nexus Objekte einzulesen und in ein *ResultSet* zu konvertieren, wird etwa 950 ms. benötigt (etwa 190ms. für 100 Objekte). Daraus ergeben sich folgende Schätzungen:

Table 6: Anteil des Einlesens und Aufbauens der ResultSets

Test	Durchschnittliche Laufzeit	Zeit um ResultSet aufzubauen
1	5575 ms.	51 Prozent
2	3310 ms.	57 Prozent
3	2530 ms.	60 Prozent
4	1379 ms.	69 Prozent
5	972 ms.	98 Prozent
6	492 ms.	77 Prozent
7	420 ms.	90 Prozent

Der Prozentuale Anteil ist besonders hoch in Test 5 und Test 7, da in diesen Tests der *ApplyRestriction* Operator verwendet wurde und nur 10 anstatt 100 Objekte im *MergeResultSet* Operator vereinigt werden mußten.

Der Prozentuale Anteil sinkt mit dem Anstieg der Anzahl der Server, da der *MergeResultSet* Operator mit steigender Zahl der zu vereinigenden *ResultSets* mehr Zeit benötigt ein weiteres *ResultSet* mit den bereits vereinigten zu vereinigen.

8 Rückblick und Ausblick

Anschließend sollen in einem Rückblick die Resultate der Diplomarbeit betrachtet werden. Dabei soll ein Ausblick auf weitere Erweiterungen der Föderation gegeben werden.

8.1 Rückblick

Der Ablauf der Anfrageverarbeitung war bisher hart-Codiert. Um die Anfrageverarbeitung in der Föderation zu flexibilisieren, wurde der Algorithmus in Operatoren implementiert. Dadurch können einzelne Teile des Algorithmus leichter ausgetauscht werden. Auch alternative Operatoren können zur Laufzeit ausgewählt werden und Operatoren können zur Laufzeit in die Anfrageverarbeitung eingefügt oder weggelassen werden.

Zuerst wurden die von der Föderation zu verwendenden Datenstrukturen vorgestellt. Dabei wurde auch die Schnittstelle, welche der einzubindende Cache implementieren soll entworfen.

Danach wurde die abzuarbeitenden Algorithmen der Föderation betrachtet und die Architektur der Föderation entworfen. Dabei wurde eine Aufteilung der Algorithmen in Operatoren vorgenommen.

Anschließend wurden im Entwurf die Algorithmen der Operatoren, bzw. deren Methoden entworfen und vorgestellt.

Im Feinentwurf wurden die Java Klassen vorgestellt, welche die Operatoren implementieren. Ebenfalls wurden die von der Föderation verwendeten Datenstrukturen und Hilfsklassen vorgestellt.

Abschließend wurde in Tests die Funktionalität und die Performance der implementierten Föderation getestet. Dabei war der Test der grundlegenden Funktionen der Föderation erfolgreich.

8.2 Ausblick

Die Föderation verwendet die Klasse *ResultSet* um *AWML* Dokumente zu repräsentieren. Diese Klasse ist allerdings nicht dazu in der Lage *Extended Class Schemas* zu verwenden. Um die Verarbeitung des *scope* Elements eines *AWML* Dokuments zu ermöglichen, sollte die Klasse erweitert werden. Dabei sollte auch bei Typ-Vergleichen unter Verwendung der *query* Methode des *ResultSets* Typ-Vergleiche mit Typen aus einem *Extended Class Schema* möglich sein.

Werden Objekte auf Servern registriert, entfernt oder aktualisiert, dann kann es vorkommen, daß nicht alle Server diese Operation erfolgreich durchführen. Wurden nicht alle Objekte erfolgreich registriert, dann werden die Objekte wieder von den Servern entfernt. Dabei ist allerdings nicht sichergestellt, daß in diesem Fall alle Objekte entfernt werden können. Auch beim entfernen und aktualisieren kann nicht garantiert werden, daß die Operation für alle oder keine Objekte durchgeführt wird.

Um die ACID Eigenschaften dieser Operationen zu sichern, ist es sinnvoll einen 2-Phasen-Commit Protokoll auf den Nexus Diensten und in der Föderation zu implementieren.

DNF-Operatoren wie *RemoveTautologies*, *RemoveDuplicateTerms MinimizeConjunctions* und *MinimizeConjunctionsSem* könnten zum Beispiel je nach der Verteilung der Daten, dem Aufbau des *RestrictionOperator*-Baums und der Selektivität der Vergleichs-Prädikate in der Reihenfolge vertauscht oder mehr als einmal in den Anfrage-Baum eingefügt werden.

Je nach Anzahl der Konjunktionen, der Auslastung des *Area Service Registers* und den verwendeten Vergleichs-Prädikaten, könnte der *GetRequestMatrix* Operator versuchen die Anzahl der Anfragen an das *Area Service Register* zu minimieren und dabei die gesamte *DNF* an die Nexus Server der Dienste-Schicht übergeben, anstatt die für die jeweiligen Server bestimmten Konjunktionen.

Nexus Server der Dienste-Schicht könnten beim *Area Service Register* zusätzlich optional eine Klasse registrieren, welche dazu verwendet werden kann, um Anfragen an diesen Server zu stellen. Die Föderation könnte dann diese Klasse von der im *Area Service Register* registrierten Lokation herunterladen und für diesen Server verwenden. Dabei sollte der Code signiert sein. Dadurch könnte jeder Nexus Server der Dienste-Schicht selbst bestimmen, wie er angesprochen wird.

9 Anhang

9.1 Konfiguration der Föderation

Die Konfiguration der Föderation erfolgt mit der Konfigurationsdatei *Federation.cfg*. Sie wird mit der Klasse *ConfigFileReader* eingelesen. Die Konfigurationsdatei hat folgenden Aufbau:

```
# The locator of the area service register.
asr=nexus:http://127.0.0.1:8081/soap/servlet/rpcrouter|urn:QueryComponent
# The URL of the standard class schema.
awsURL=http://127.0.0.1:8081/aws-0.1.xsd
# The class that will send queries to the servers.
serverClass=de.uni_stuttgart.nexus.federation.servers.ReadFiles
# The class of the cache object
cacheClass=de.uni_stuttgart.nexus.federation.servers.Cache
# The class of the object that sends queries to the area service register
asrClass=de.uni_stuttgart.nexus.federation.servers.ASRFile
# Should federation pass the whole DNF to the servers ?
passDNF=false
```

Der Defaultwert für den Locator des ASR ist *nexus:http://127.0.0.1:8080/soap/servlet/rpcrouter|urn:QueryComponent.*

Der Defaultwert für die URL des AWS Schemas ist *http://127.0.0.1:8080/aws-0.0.xsd*.

Die Klassen, welche mit den Nexus Servern kommuniziert, wird mit *serverClass* angegeben. Die Klasse des Cache wird mit *cacheClass* angegeben. Beide Klassen müssen das *ServerInterface* Interface implementieren.

Die Klasse, welche mit dem *Area Service Register* kommuniziert, wird mit *asrClass* angegeben. Die Klasse muß das *ASRQueryInterface* implementieren.

Mit *passDNF* wird angegeben, ob die Föderation beim Bestimmen der Zielserver die Anzahl der Anfragen an das *Area Service Register* minimieren soll, dafür allerdings die komplette Anfrage (in *DNF*) an die Nexus Server übergeben soll, anstatt nur die Konjunktionen, für welche der Server Nexus Objekte liefern könnte zu übergeben.

10 Literaturliste

[ArcSDE]
http://www.dnr.state.ak.us/lris/gis/documentation/internal/vendor/ArcSDE_documents/arcsde_dev_help/sql_interface/concepts/understanding_spatial_relations.htm

[Apache SOAP]
Apache SOAP Dokumentation
http://xml.apache.org/soap/

[Apache Tomcat]
Apache Tomcat 4.0 Dokumentation
http://jakarta.apache.org/tomcat/

[Enge 2003]
Effiziente Repräsentation semistrukturierter Ergebnismengen im Haupspeicher
Johannes Enge

[IBMDB2]
IBM DB2 Spatial Extender User's Guide and Reference - Version 8

[Liang 2002]
Shan Liang: Konzeption und Implementierung der Förderationskomponente in Nexus

[Nicklas et al 2001]
A Model-Based, Open Architecture for Mobile, Spatially Aware Applications
Daniela Nicklas, Matthias Großmann, Thomas Schwarz, Steffen Volz und Bernhard Mitschang
Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD2001) - 2001

[Nicklas et al 2003]
D. Nicklas et al: Information Management and Exchange in neXus - internal Technical Report - v1.5, Universität Stuttgart 2003

[OpenGIS]
OpenGIS Simple Features Specification for CORBA Revision 1.0
http://www.opengis.org/docs/99-054.pdf

[Schöning 1995]
Logik für Informatiker von Uwe Schöning

Erklärung

Ich versichere, daß ich die Arbeit selbstständig verfasst
und nur die angegebenen Hilfsmittel verwendet habe.

_____

(Steffen Motzer)