Humboldt-Universität zu Berlin

Institut für Informatik

Grundlagen regelbasierter Anfrageoptimierung für SPARQL

Diplomarbeit von Olaf Hartig

September 2006

1. Gutachter: Prof. Johann Christoph Freytag

2. Gutachter: Prof. Dr. Felix Naumann

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig angefertigt habe. Es wurden nur die in der Arbeit genannten Quellen benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches gekennzeichnet.

Berlin, den 26. September 2006

Olaf Hartig

Danksagung

An erster Stelle möchte ich meinem Betreuer Ralf Heese danken. Von den Anfängen der Themenfindung bis zum Tag der Abgabe hat er mir sowohl in inhaltlichen als auch stilistischen Fragen immer beratend zur Seite gestanden. Sehr herzlich möchte möchte ich mich auch bei Herrn Freytag für die konstruktive Kritik vor der Abgabe dieser Arbeit bedanken.

Olaf Hartig

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ziele der Arbeit . . . 1
1.2 Gliederung der Arbeit . . . 2

2 Grundlagen **4**
2.1 Das Semantic Web . . . 4
2.2 Das Resource Description Framework . . . 4
2.3 SPARQL . . . 7
2.3.1 Einführung . . . 7
2.3.2 Grundlegendes zur Syntax . . . 9
2.3.3 Einfache Graphmuster . . . 10
2.3.4 Optionale Informationen . . . 10
2.3.5 Alternativen . . . 11
2.3.6 Wertebeschränkungen . . . 12
2.3.7 Datenmengen und benannte Graphen . . . 12
2.3.8 Anpassung der Lösungsmenge . . . 14
2.4 Bearbeitung von Anfragen . . . 15
2.4.1 Anfragebearbeitung in DBMS . . . 15
2.4.2 Forschungsstand der Bearbeitung von SPARQL-Anfragen . . . 17

3 Das SPARQL Query Graph Model **20**
3.1 Formale Definition des SQGM . . . 20
3.1.1 Grundlegende Definitionen . . . 20
3.1.2 Klassen von Operatoren . . . 22
3.1.3 Repräsentation von RDF-Graphen . . . 23
3.1.4 Repräsentation von einfachen Graphmustern und Wertebeschränkungen . . . 24
3.1.5 Repräsentation von Alternativen . . . 25
3.1.6 Repräsentation gruppierter Graphmuster . . . 25
3.1.7 Repräsentation optionaler Graphmuster . . . 26
3.1.8 Repräsentation von Lösungsanpassungen . . . 26
3.1.9 Operatoren zur Ergebniserzeugung . . . 26
3.1.10 Repräsentation von RDF-Graph-Klauseln und benannten Graphen . . . 28
3.1.11 Weitere Details . . . 28
3.1.12 Zusammenfassung . . . 29
3.2 Grafische Darstellung eines SQGM . . . 31
3.3 Anfragekompilierung auf Basis des SQGM . . . 33
3.3.1 Anfrageübersetzung und -vereinfachung . . . 35
3.3.2 Logische Optimierung . . . 35
3.3.3 Physische Optimierung mit kostenbasierter Auswahl . . . 36

4 Anfrageübersetzung und -vereinfachung **38**
4.1 Erstellung eines SQGM . . . 38
4.1.1 Modellierung der RDF-Datenmenge . . . 39
4.1.2 Modellierung des Graphmusters . . . 41
4.1.3 Modellierung von Lösungsanpassungen . . . 47
4.1.4 Modellierung der Ergebnisform . . . 48
4.1.5 Eigenschaften des modellierten SQGM . . . 50
4.2 Normalisierung . . . 51
4.2.1 Eingeschränkte Ausgänge . . . 52
4.2.2 Lokalität von Variablenbindungen . . . 53
4.2.3 Normalisierte Wertebeschränkungen . . . 56
4.2.4 Freiheit von leeren Graphmuster-Operatoren . . . 59
4.2.5 Vergleich mit anderen Normalformen . . . 62
4.3 Vereinfachung . . . 62
4.3.1 Entfernung von Redundanzen . . . 62
4.3.2 Behandlung von Widersprüchen . . . 65

5 Implementierung der Konzepte **71**
5.1 Die Basis – Jena und ARQ . . . 71
5.2 Implementierung des SQGM . . . 73
5.3 Einbettung des SQGM in Jenas Anfragebearbeitung . . . 75
5.4 Implementierung des Systems zur Regelausführung . . . 78
5.4.1 Anforderungen an das Regelausführungssystem . . . 80
5.4.2 Entwurfsmodell des Regelausführungssystems . . . 80
5.4.3 Implementierung der Normalisierung eines SQGM . . . 83
5.4.4 Implementierung der Vereinfachung eines SQGM . . . 85
5.5 Implementierung der regelbasierten Restrukturierung . . . 85
5.5.1 Anforderungen an die Komponente zur Restrukturierung . . . 86
5.5.2 Entwurfsmodell des Restrukturierungssystems . . . 87
5.6 Implementierung einer Testumgebung . . . 88

6 Heuristiken für regelbasierte Optimierung **90**
6.1 Grundlegende Eigenschaften von Heuristiken . . . 90
6.2 Die Testumgebung . . . 91
6.3 ARQ's Fast-Path-Algorithmus für geeignete Tripelmuster . . . 92
6.3.1 Zugrundliegende Vermutung . . . 92
6.3.2 Notwendige Restrukturierungsregeln . . . 93
6.3.3 Notwendige Transformationsregeln . . . 97
6.3.4 Spezifikation der Heuristik . . . 103
6.3.5 Experimentelle Untersuchung . . . 104
6.3.6 Evaluierung . . . 108
6.4 ARQ's Fast-Path-Algorithmus für alle Tripelmuster . . . 108
6.4.1 Experimentelle Untersuchung . . . 109
6.4.2 Evaluierung . . . 109
6.5 ARQ's Fast-Path-Algorithmus für beschränkte Tripelmuster . . . 110
6.5.1 Experimentelle Untersuchung . . . 110
6.5.2 Evaluierung . . . 111
6.6 ARQ's Fast-Path-Algorithmus bei vereinigten Graphmustern . . . 113
6.6.1 Zugrundliegende Vermutung . . . 113
6.6.2 Notwendige Transformationsregeln . . . 114
6.6.3 Notwendige Restrukturierungsregeln . . . 117
6.6.4 Spezifikation der Heuristik . . . 117
6.6.5 Experimentelle Untersuchung . . . 118
6.6.6 Evaluierung . . . 118

7 Zusammenfassung und Ausblick **120**

A Testanfragen **124**

Abkürzungsverzeichnis **129**

Abbildungsverzeichnis

2.1 Ein RDF-Graph zur Repräsentation der Aussagen aus Beispiel 2.1 und 2.2 6
2.2 Ein RDF-Graph mit leeren Knoten 6
2.3 Hauptbestandteile der Anfragebearbeitungseinheit eines DBMS nach [GMWU02] . 16

3.1 SQGM für eine Auswahl-Anfrage mit optionalem Graphmuster, Lösungsmengenanpassung und zusammengesetztem Default-Graph 32
3.2 Ein SQGM mit CONSTRUCT-Box für eine Anfrage mit RDF-Graph-Klausel und Wertebeschränkung 33
3.3 Schritte bei der Anfragekompilierung in Anlehnung an [GS05] 33
3.4 Teilschritte der Anfrageübersetzung und -vereinfachung 35

4.1 Rekursiver Algorithmus zur Erzeugung eines SQGM-Operators für ein Graphmuster 42
4.2 Ein links-assoziativer Verbund-Baum für eine Gruppe von vier Graphmustern . . . 44
4.3 Grafische Darstellung des SQGM mit den Operatoren aus den Beispielen 4.1, 4.2, 4.3 und 4.4 51
4.4 Ebenen und Schichten eines SQGM 52
4.5 Normalisierte Form des SQGM aus Abbildung 4.3 56

5.1 Die Jena-Architektur nach [CDD+03] 72
5.2 Entwurfsmodell der Klassen für Operatoren und Datenflüsse eines SQGM 74
5.3 Entwurfsmodell einer SQGM-Repräsentation von SPARQL-Anfragen 75
5.4 Entwurfsmodell für die Einbettung einer alternativen Anfragebearbeitungskomponente für ARQ auf Basis des SQGM 76
5.5 Entwurfsmodell zur Verdeutlichung der Beziehung zwischen der ARQ-Repräsentation von Anfragen und der Repräsentation auf Basis des SQGM 76
5.6 Entwurfsmodell zur Erzeugung von SQGM-Repräsentationen für Anfragen aus der ARQ-Repräsentationsform 77
5.7 Entwurfsmodell der prototypischen Anfragebearbeitungskomponente als Erweiterung der ARQ-Realisierung 77
5.8 Anfragebearbeitung mit erweiterter Anfragebearbeitungskomponente 78
5.9 Ein SQGM vor und nach der Ausführung einer RemoveEqualTriplePatterns-Regelinstanz 79
5.10 Entwurfsmodell der Klassen des Regelausführungssystems 81
5.11 Algorithmus zur Abarbeitung einer Menge von Regelinstanzen 82
5.12 Algorithmus zum Übernehmen von Folgeregeln in die Abarbeitungsliste 82
5.13 Entwurfsmodell der Regel SplitConstraint 83
5.14 Entwurfsmodell der Regelsystems zur Normalisierung eines SQGM 84
5.15 Algorithmus zur Normalisierung eines SQGM unter Verwendung des Regelausführungssystems 84
5.16 Entwurfsmodell der Regelsystems zur Vereinfachung eines normalisierten SQGM . 85
5.17 Algorithmus zur Vereinfachung eines SQGM unter Verwendung des Regelausführungssystems 85

5.18 Entwurfsmodell der Klassen des Restrukturierungssystems 87
5.19 Algorithmus zur Abarbeitung einer Menge von Restrukturierungsregeln 87
5.20 Entwurfsmodell der wichtigsten Klassen für die Testumgebung 88

6.1 Algorithmus zum Finden von zwei verschmelzbaren Graphmuster-Operatoren . . . 95
6.2 Rekursiver Algorithmus zum Finden aller Graphmuster-Operatoren, die über einen Verbund-Baum verbunden sind und für eine Verschmelzung in Frage kommen . . . 96
6.3 Algorithmus zum Erzeugen einer Sequenz von Regelinstanzen, um zwei verbundene Graphmuster-Operatoren zu verschmelzen 96
6.4 Anwendung der Transformationsregel MergeJoinedGPOs auf ein SQGM 98
6.5 Beispielhaft angedeuteter Verbund-Baum bei der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Beginn) 101
6.6 Beispielhaft angedeuteter Verbund-Baum während der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Fortsetzung) 102
6.7 Beispielhaft angedeuteter Verbund-Baum während der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Abschluss) 102
6.8 Verfeinerter Algorithmus zum Erzeugen einer Sequenz von Regelinstanzen, um zwei verbundene Graphmuster-Operatoren zu verschmelzen 104

Kapitel 1

Einleitung

Das World Wide Web ist die größte Sammlung von elektronisch gespeicherten Informationen und Daten. In seiner heutigen Form können diese Informationen aber nur von Menschen verstanden werden. Es gibt jedoch viele mögliche Anwendungen für die es notwendig wäre, dass diese Informationen auch von Maschinen interpretiert und verarbeitet werden können. Das ist zum Beispiel eine semantische Suche oder die automatische Aushandlung von Terminen. Um diesen Misstand zu beheben wird das Semantic Web entwickelt. Das Ziel dieser Entwicklung ist es, die nach syntaktischen Regeln gespeicherten Informationen und Daten um semantische Informationen zu erweitern und diese damit für Maschinen verständlich zu machen.

Eine Grundlage des Semantic Web bildet das vom World Wide Web Consortium (W3C) entwickelte Resource Description Framework (RDF) [KC04]. Diese Spezifikation definiert grundlegende Mechanismen zur Beschreibung von Ressourcen und deren Beziehungen, welche als Graph dargestellt werden können. Diese Beschreibungen sind unter Umständen in verschiedenen, möglicherweise verteilten RDF-Datenmengen hinterlegt. Um gezielt Informationen aus diesen Mengen abfragen zu können, wird vom W3C momentan die Anfragesprache SPARQL [PS06] definiert. Diese deklarative Sprache beruht auf dem Finden von Teilgraphen anhand eines Musters.

Wegen des geringen Alters der SPARQL-Spezifikation existieren derzeit zwar bereits einige Systeme zur Bearbeitung entsprechender Anfragen, jedoch gibt es nur wenige Publikationen zu dieser Thematik. Da die verschiedenen Systeme momentan auf vergleichsweise naiven Ansätzen zur Ermittlung der Anfrageergebnisse beruhen, besteht hier insbesondere die Notwendigkeit der Entwicklung von Konzepten für eine effizientere Anfrageausführung. Zur Motivation sei an dieser Stelle auf die Diskussion der Bearbeitungszeit einer SPARQL-Anfrage in einem Online-Forum[1] verwiesen. Das dort vorgeschlagene Vertauschen der Position einiger Anfrageelemente hat eine Reduzierung der Bearbeitungszeit auf weniger als 0,5% ermöglicht. Die Aufgabe einer entsprechenden Komponente zur Anfrageoptimierung müsste solche Fälle erkennen und automatisch die betreffenden Elemente in eine möglichst optimale Reihenfolge bringen. Wegen der Parallelen ist es in diesem Zusammenhang möglich, die auf dem Gebiet der Anfrageoptimierung im Datenbankbereich entwickelten Prozesse auf SPARQL zu übertragen. Die tatsächliche Ausgestaltung dieser Prozesse erfordert jedoch die Entwicklung und Untersuchung von angepassten oder neuen Konzepten, welche speziell auf SPARQL bezogen sind.

1.1 Ziele der Arbeit

Die Arbeit soll einen Beitrag zur konzeptuellen Auseinandersetzung mit der Anfragebearbeitung für SPARQL leisten. Es sollen grundlegende Konzepte für die auf heuristischen Regeln basierende Optimierung der Ausführung von SPARQL-Anfragen entwickelt werden. Da dieser Optimierungsschritt auf Transformationsregeln zur Umformung der Anfragen aufbaut, bilden die Entwicklung

[1] http://tech.groups.yahoo.com/group/jena-dev/message/21436

eines Modells zur Anfragerepräsentation und die Umformungsregeln die theoretischen Schwerpunkte dieser Arbeit. Sie werden ergänzt durch die praktische Umsetzung der entwickelten Konzepte und die experimentelle Untersuchung einiger Beispielheuristiken.

Im Rahmen diese Arbeit ist zunächst ein für die weiteren Betrachtungen geeignetes Repräsentationsmodell für SPARQL-Anfragen zu entwickeln, welches an das Query Graph Model (QGM) für Datenbankanfragen des Starburst-Projektes [HCL+90] angelehnt ist. Da es in dieser Arbeit der regelbasierten Optimierung als grundlegende Repräsentationsform der SPARQL-Anfragen dient, muss es die hierfür notwendigen Informationen beinhalten. Perspektivisch sollte es jedoch möglich sein, dieses Modell im gesamten Prozess der Anfragebearbeitung einzusetzen. Aus diesem Grund muss es so entworfen werden, dass es die Semantik der Anfragen vollständig repräsentiert und bezüglich zusätzlich benötigter Informationen erweiterbar ist.

Ein weiterer grundlegender Schritt ist die Entwicklung von Konzepten, die es ermöglichen das entwickelte Repräsentationsmodell im Rahmen einer regelbasierten Optimierung einzusetzen. Die fundamentale Idee dieser Variante der Anfrageoptimierung ist die Restrukturierung der Anfragen auf Basis von Heuristiken. Deshalb ist es hierfür unter anderem notwendig, Möglichkeiten für eine entsprechende Anfrageumformung zu entwerfen.

Die theoretischen Überlegungen sollten in der Praxis getestet werden. Zu diesem Zweck müssen die in der Diplomarbeit erarbeiteten Konzepte zunächst implementiert und dann in die Anfragebearbeitung des Anfragesystems ARQ[2] [Sea06] integriert werden. Mit Hilfe der theoretischen Grundlagen können letztendlich einige mögliche Heuristiken für die regelbasierte Optimierung formuliert und auf Basis der implementierten Konzepte experimentell untersucht werden.

1.2 Gliederung der Arbeit

Das Vorgehen bei der Entwicklung der Konzepte, deren Umsetzung und Untersuchung wurde in vier Schritte unterteilt. Zunächst wurde das notwendige Repräsentationsmodell für SPARQL-Anfragen entworfen und spezifiziert. Danach wurden die theoretischen Grundlagen für eine regelbasierte Umformung der Anfragen und eine darauf aufbauende Optimierung entwickelt. Dann wurden Modelle zur Umsetzung dieser Konzepte entworfen und implementiert und letztendlich wurden die umgesetzten Ideen in einem möglichen Anwendungsszenario eingesetzt und untersucht. An dieser Unterteilung orientiert sich auch die Gliederung der vorliegenden Diplomarbeit.

Das Kapitel 2 stellt zunächst die wichtigsten Ideen und Technologien vor, auf denen die in dieser Arbeit entwickelten Konzepte aufbauen. Ein Hauptaugenmerk wurde hierbei auf die RDF- und SPARQL-Spezifikationen gelegt, da diese die entscheidenden Grundlagen bilden und in diesem Rahmen eingeführte Begriffe in der späteren Arbeit immer wieder verwendet werden. Zum besseren Verständnis werden die jeweiligen Beschreibungen durch viele Beispiel ergänzt. Neben den Spezifikationen wird auch die oben bereits erwähnte Anfragebearbeitung in Datenbanksystemen grob beschrieben, da die grundlegendsten, der in diesem Zusammenhang eingeführten Konzepte auch für die Bearbeitung von SPARQL-Anfragen übernommen wurden und die im Folgenden entwickelten Ideen kennzeichnet. Abgerundet wird das Kapitel mit einer kurzen Erwähnung anderer Arbeiten, welche sich in unterschiedlichster Weise mit der Bearbeitung von Anfragen an RDF-Datenmengen und SPARQL beschäftigen.

Das dritte Kapitel dient der Vorstellung des SPARQL Query Graph Model (SQGM), welches die oben erwähnte Repräsentation von SPARQL-Anfragen darstellt. In dem Kapitel wird das SQGM zunächst formal definiert. Neben den RDF- und SPARQL-Spezifikationen bildet diese Definition eine der essentiellen Grundlagen dieser Arbeit, da sich alle weiteren in der Arbeit entwickelten Konzepte auf dieses Modell beziehen. Der Definition folgt die Spezifikation einer Möglichkeit zur Visualisierung eines SQGM. Um die Verwendung des Modells zu motivieren und die entwickelten Ideen einordnen zu können, wird das Kapitel durch die Beschreibung eines möglichen Vorgehens bei der Bearbeitung von SPARQL-Anfragen auf Basis des SQGM komplettiert.

Im vierten Kapitel werden die Grundlagen entwickelt, welche es erlauben, aus einer SPARQL-Anfrage ein SQGM zu erzeugen und dieses in eine Form zu bringen, welche für die spätere Optimie-

[2] `http://jena.sourceforge.net/ARQ/`

rung verwendet werden kann. Dabei kommen Regeln zur Umformung von SQGMs zum Einsatz. Diese bilden bereits die Basis der regelbasierten Optimierung.

Eines der Ziele dieser Arbeit ist die praktische Untersuchung der in dieser Arbeit entwickelten Konzepte, welche zu diesem Zweck implementiert werden mussten. Das fünfte Kapitel beschreibt diese Umsetzung. Ein Schwerpunkt liegt hier auf dem entworfenen System zur Ausführung von Regeln zur Umformung von SQGMs. In diesem Zusammenhang wurden weitere Grundlagen für die regelbasierten Optimierung entwickelt, welche hier definiert sind.

Das Kapitel 6 dient der bespielhaften Untersuchung einiger Heuristiken, welche für eine Anfrageoptimierung eingesetzt werden könnten. Zunächst wird hier ein spezielles Anwendungsszenario vorgestellt, auf das die Beispielheuristiken zugeschnitten sind. Danach wird jede Heuristik motiviert, die benötigten Regeln und Algorithmen diskutiert und eine entsprechende experimentelle Untersuchung beschrieben und ausführlich ausgewertet.

Das abschließende, siebte Kapitel gibt eine kurze Zusammenfassung der neuen Konzepte und gewonnenen Erkenntnisseund enthält einige mögliche Anknüpfungspunkte und Anregungen zur Weiterentwicklung der im Rahmen der Arbeit entstandenen Ergebnisse.

Kapitel 2

Grundlagen

Eine Reihe von Ideen, Technologien und Begriffen bilden die Grundlagen, auf denen die im Rahmen dieser Arbeit entwickelten Konzepte und deren Umsetzung aufbauen. Dieses Kapitel wird diese Grundlagen vorstellen und, soweit es für das Verständnis dieser Arbeit notwendig ist, detailiert beschreiben. Bei wichtigen Begriffen wird auf deren Definition in den entsprechenden Spezifikationen verwiesen.

Der Vision des Semantic Web, welche als Motivation dieser Arbeit gelten kann, ist der erste Abschnitt gewidmet. In den folgenden beiden Abschnitten wird das Resource Description Framework zur Wissensrepräsentation und die dafür entwickelte Anfragesprache SPARQL vorgestellt. Der letzte Abschnitt beschäftigt sich schließlich mit der Ausführung von Anfragen.

2.1 Das Semantic Web

Das World Wide Web ist für viele Menschen als Informationsquelle und als Plattform zur Kommunikation heutzutage unerlässlich. Es ist die grösste Sammlung von elektronisch gespeicherten Informationen und Daten. In seiner momentanen Form können diese Informationen aber nur von Menschen verstanden werden. Wäre es möglich, dass diese Menge an Informationen und Daten auch von Maschinen interpretiert und verarbeitet werden kann, so würde das die Möglichkeit für viele neue Anwendungen eröffnen. Im einfachsten Fall wäre das eine semantische Suche. Die Möglichkeiten gehen aber weit darüber hinaus, wie beispielsweise die Vision einer automatischen Aushandlung und Abstimmung von Terminen zwischen mehreren Software-Agenten, welche auf unterschiedlichen Datenbasen operieren [BLHL01].

Das Ziel des Semantic Web ist es Standards und Technologien zu entwickeln, welche es Maschinen ermöglichen die Informationen des Web zu verstehen [KM01]. Hierfür müssen die nach syntaktischen Regeln gespeicherten Informationen und Daten um semantische Informationen erweitert werden, welche mit dem im folgenden Abschnitt vorgestellten Resource Description Framework (RDF) beschrieben werden können.

2.2 Das Resource Description Framework

Dieser Abschnitt beschreibt das Resource Description Framework (RDF) und stellt die wichtigsten, in diesem Zusammenhang auftretenden Begriffe vor. Für eine ausführlichere Einführung sei auf [MM04] verwiesen.

RDF ist eine Sprache zur Repräsentation von Wissen und eine der wichtigsten Grundlagen des Semantic Web. Sie wurde vom W3C im Rahmen seiner Semantic-Web-Aktivitäten[1] spezifiziert. Diese Spezifikation umfasst die Definition einer abstrakten Syntax [KC04] und einer formalen Semantik [Hay04]. Sie wird durch RDF-Schema [BG04], eine RDF-Anwendung zur Definition von

[1] `http://www.w3.org/2001/sw/`

RDF-Vokabularen, ergänzt. Auf Basis der abstrakten Syntax haben sich verschiedene Möglichkeiten zur Darstellung von RDF-Datenmengen entwickelt. Die wichtigsten sind die einfache, zeilenweise Textdarstellung Turtle [Bec04b], die, unter anderem in [MM04] verwendete Darstellung in Form eines Graphen und eine vom W3C spezifizierte XML-Syntax [Bec04a].

Die Grundidee von RDF ist die Beschreibung von Ressourcen durch eine Reihe von Eigenschaften und zugehörigen Werten. Unter *Ressourcen* werden im Rahmen von RDF Entitäten verstanden. Ressourcen sind demzufolge konkrete und abstrakte Gegenstände über die Aussagen getroffen werden können und die miteinander in Beziehung stehen. Beispielsweise sind Internetseiten, Passagen aus Dokumente, Personen oder Bücher Ressourcen. Eine Aussage über eine Ressource ist ebenfalls eine Ressource im Sinne von RDF. Wie bereits erwähnt besteht jede Aussage aus der beschriebenen Ressource, einer Eigenschaft und einem Eigenschaftswert.

Beispiel 2.1 *Über den Berg Mont Blanc kann folgende Aussage getroffen werden.*

«Der Mont Blanc hat eine Höhe von 4810 Metern.»

Die in dieser Aussage beschriebene Ressource ist der Berg Mont Blanc. *Die Eigenschaft ist die* Höhe *und deren Wert ist* 4810 *Meter.* □

Sowohl die Eigenschaften, als auch deren Werte können weitere Ressourcen sein, welche in anderen Aussagen beschrieben werden. Eigenschaftswerte können auch konstante Werte, als *Literal* bezeichnet, sein.

Beispiel 2.2 *Die folgenden Aussagen können getroffen werden.*

«Der Mont Blanc ist ein Teil der Alpen.»
«Die Alpen befinden sich in Europa.»
«Der Mont Blanc hat den Namen „Mont Blanc".»

Die Ressource Alpen *tritt in der ersten Aussage als Eigenschaftswert auf und wird in der zweiten Aussage näher beschrieben. Der Wert in der dritten Aussage ist ein Literal.* □

Um die Ressourcen eindeutig zu identifizieren, werden in RDF *Uniform Resource Identifiers* (URIs) [BLFM98] verwendet. Lange URIs können durch die Verwendung von qualifizierten Namen, im Folgenden als *QName* bezeichnet, abgekürzt werden. Ein QName besteht aus einem *Präfix* und einem *lokalen Namen*, verknüpft durch einen Doppelpunkt. Jeder Präfix repräsentiert einen Namensraum für URIs.

Beispiel 2.3 *Der Berg Mont Blanc könnte durch die URI*

`http://www.example.org/mountains/MontBlanc`

identifiziert werden. Wenn der Namensraum `http://www.example.org/mountains/` *dem Präfix* `exm` *zugewiesen ist, dann entspricht der QName* `exm:MontBlanc` *der repräsentierenden URI.* □

Jede Aussage wird in RDF durch ein *Tripel* [KC04, 6.1 RDF Triples] repräsentiert. Tripel bestehen aus den drei Element *Subjekt*, *Prädikat* und *Objekt*. Dabei ist das Subjekt die beschriebene Ressource, das Prädikat die Eigenschaft und das Objekt der Wert der Eigenschaft.

Beispiel 2.4 *Die Aussage aus Beispiel 2.1 wird durch ein Tripel mit dem Subjekt*

`http://www.example.org/mountains/MontBlanc`

dem Prädikat

`http://www.example.org/mountains/height`

und dem Objekt

`4810`

dargestellt, wobei die URI `http://www.example.org/mountains/height` *das Konzept der Höhe eines Berges repräsentiert. Unter Verwendung des Präfix* `exm` *aus dem Beispiel 2.3 kann die Schreibweise wie folgt abgekürzt werden. Das Subjekt des Tripels ist* `exm:MontBlanc`*, das Prädikat* `exm:height` *und das Objekt der konstante Wert 4810.* □

Eine Menge von Tripeln kann durch einen gerichteten Graphen repräsentiert werden, der als *RDF-Graph* [KC04, 6.2 RDF Graph] bezeichnet wird. Dabei bilden die Subjekte und Objekte die Knoten und die Prädikate bilden die Kanten, welche die jeweils entsprechenden Subjekt-Knoten mit den Objekt-Knoten verbinden. In einer grafischen Darstellung von RDF-Graphen werden die Knoten für Ressourcen durch Ovale und die Knoten für Literale durch Rechtecke gekennzeichnet. Die jeweiligen Elemente des Graphen werden mit der URI bzw. dem Literal des jeweils zugehörigen Tripel-Elements beschriftet.

Beispiel 2.5 *Die in den Beispielen 2.1 und 2.2 getroffenen Aussagen können durch den Graphen in Abbildung 2.1 repräsentiert werden.*

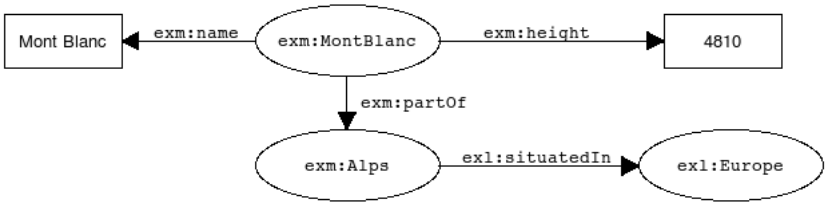


Abbildung 2.1: Ein RDF-Graph zur Repräsentation der Aussagen aus Beispiel 2.1 und 2.2 □

Es sei an dieser Stelle darauf hingewiesen, dass die Begriffe *RDF-Graph* und *Tripelmenge* synonym verwendet werden können. Letztendlich repräsentieren beide die Menge der Aussagen, welche durch die dargestellten bzw. enthaltenen Tripel formuliert werden.

Ein weiteres Konzept von RDF sind *leere Knoten* (engl.: blank nodes) [KC04, 6.6 Blank Nodes]. Sie repräsentieren anonyme Ressourcen denen keine URI zugeordnet ist.

Beispiel 2.6 *Aus der Aussage*

> *«Der Mont Blanc wurde erstmals am 8.August 1786 von Jacques Balmat und Michel Paccard bestiegen.»*

kann der in Abbildung 2.2 dargestellte RDF-Graph modelliert werden. Das nicht weiter identifizierte Konzept der Erstbesteigung des Mont Blanc und die beiden Personen werden dabei durch jeweils einen leeren Knoten repräsentiert.

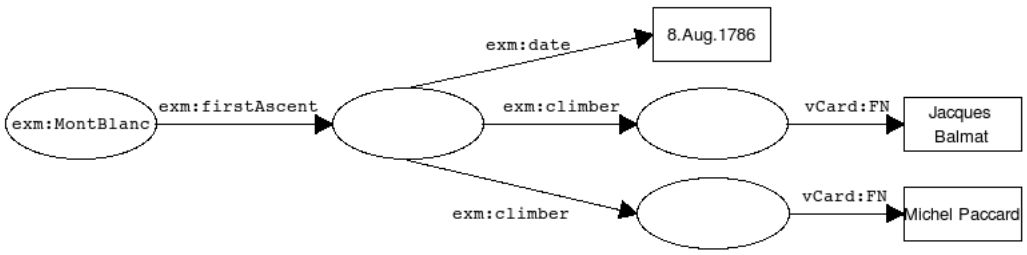


Abbildung 2.2: Ein RDF-Graph mit leeren Knoten □

Weil es nicht immer möglich oder sinnvoll ist einen Graphen zur Repräsentation von RDF-Daten zu zeichnen, wurden einfache Textdarstellungen entworfen, welche auf der Idee beruhen, die einzelnen Tripel zeilenweise formal zu notieren. Eine dieser Textdarstellungen ist die in [Bec04b] definierte Turtle-Notation, bei der die Tripel üblicherweise einzeln notiert und durch einen Punkt abgeschlossen werden. URIs und QNamen, im Folgenden zusammen als *URI-Referenzen* bezeichnet, müssen durch die Zeichen $<$ und $>$ geklammert sein.

Beispiel 2.7 *Der in Abbildung 2.1 dargestellte RDF-Graph kann in Turtle wie folgt notiert werden.*

```
@prefix exm: <http://www.example.org/mountains/> .
@prefix exl: <http://www.example.org/locations/> .

exm:MontBlanc exm:name       "Mont Blanc" .
exm:MontBlanc exm:height     4810 .
exm:MontBlanc exm:partOf     exm:Alps .
exm:Alps      exl:situatedIn exl:Europe .
```
□

Eine verkürzte Schreibweise erlaubt das Zusammenfassen mehrerer Tripel mit gleichem Subjekt oder gleichem Subjekt und Prädikat.

Beispiel 2.8 *Eine abgekürzte Turtle-Darstellung des Beispiels 2.7 ist die Folgende.*

```
@prefix exm: <http://www.example.org/mountains/> .
@prefix exl: <http://www.example.org/locations/> .

exm:MontBlanc exm:name "Mont Blanc" ;
              exm:height 4810 ;
              exm:partOf exm:Alps .
exm:Alps exl:situatedIn exl:Europe .
```
□

Zur Notation von leeren Knoten werden eckige Klammern verwendet.

Beispiel 2.9 *Der RDF-Graph in Abbildung 2.2 enthält leere Knoten und kann in Turtle wie folgt notiert werden.*

```
@prefix exm:   <http://www.example.org/mountains/> .
@prefix vCard: <http://www.w3.org/2001/vcard-rdf/3.0#> .

exm:MontBlanc exm:firstAscent [ exm:date "8.Aug.1786" ;
                                exm:climber [vCard:FN "Jacques Balmat"] ;
                                exm:climber [vCard:FN "Michel Paccard"]   ].
```
□

2.3 SPARQL

2.3.1 Einführung

Der Erfolg des Semantic Web im Allgemeinen und von RDF im Speziellen ist abhängig von der Möglichkeit, auf eine standardisierte Weise auf RDF-Daten zugreifen und Anfragen an RDF-Datenmengen stellen zu können. Mit einem verbreiteteren Einsatz von RDF wurden deshalb verschiedenste Anfragesprachen entwickelt. In [HBEV04] und [MKC+02] werden diese Sprachen, welche auf unterschiedlichster Syntax und Semantik basieren, verglichen. Letztendlich konnte sich keine davon durchsetzen. Um die Entwicklung eines einheitlichen Standards voranzutreiben, hat das W3C im Februar 2004 die RDF Data Access Working Group[2] gegründet. Diese hat in [Cla05] zunächst eine Reihe von Anwendungsfällen für den Zugriff auf RDF-Daten und Anforderungen an eine RDF-Anfragesprache formuliert. Auf dieser Basis wird gerade

- die Anfragesprache SPARQL [PS06],
- ein Protokoll zur Übermittlung von SPARQL-Anfragen und Resultaten bei der Kommunikation mit Diensten zur Anfragebearbeitung [Cla06] und
- ein XML-Format zur Beschreibung von Ergebnissen einer Anfrageausführung [Bec06]

[2] http://www.w3.org/2001/sw/DataAccess/

spezifiziert. Alle drei Spezifikationen befinden sich zu diesem Zeitpunkt im Zustand einer Candidate Recommendation. Einen guten Überblick über die Bedeutung von SPARQL im Rahmen des Semantic Web gibt Dodds in [Dod06].

SPARQL ist eine deklarative Anfragesprache. Sie erlaubt es Informationen in Form von URIs, leeren Knoten und Literalen aus RDF-Graphen zu erhalten, Teilgraphen zu erfragen und neue RDF-Graphen auf Basis der abgefragten Informationen zu erstellen. Die Ausführung der Anfragen basiert auf der Idee der *Musterübereinstimmung* (engl.: pattern matching). Hierzu werden in einer Anfrage *Graphmuster* (engl.: graph pattern) beschrieben, welche Variablen enthalten können. Zu diesen Mustern werden bei der Anfrageausführung übereinstimmende Teilgraphen aus dem abgefragten RDF-Graph bestimmt und die Variablen an die entsprechenden URIs, Literale und leeren Knoten gebunden. Diese gebundenen Variablen bilden eine Lösung.

Beispiel 2.10 *Eine einfache SPARQL-Anfrage, die nach einer Ressource und dem zugehörigen Namen fragt, könnte wie folgt aussehen.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?m ?n
WHERE { ?m exm:name ?n . }
```

Wenn diese Anfrage auf dem in Abbildung 2.1 dargestellten RDF-Graph (siehe Beispiel 2.5) ausgeführt wird, erhält man das folgende Resultat.

?m	?n
http://www.example.org/mountains/MontBlanc	"Mont Blanc"

□

Da ein untersuchter RDF-Graph mehrere Übereinstimmungen mit den Graphmustern haben kann, könnte es mehrere Lösungen geben.

Beispiel 2.11 *Die folgende Turtle-Repräsentation stellt einen etwas umfangreicheren RDF-Graph dar. Dieser beinhaltet Informationen über einige Berge der Alpen.*

```
@prefix exm: <http://www.example.org/mountains/> .
@prefix exl: <http://www.example.org/locations/> .
exm:MontBlanc exm:name "Mont Blanc";
              exm:range exm:PennineAlps;
              exm:height 4810;
              exm:location exl:France , exl:Italy;
              exm:firstAscent [ exm:year 1786 ] .
exm:DufSptz exm:name "Dufourspitze";
            exm:range exm:PennineAlps;
            exm:height 4634;
            exm:location exl:Switzerland , exl:Italy;
            exm:firstAscent [ exm:year 1855 ] .
exm:FstrHorn exm:name "Finsteraarhorn";
             exm:range exm:BerneseAlps;
             exm:height 4274;
             exm:location exl:Switzerland;
             exm:firstAscent [ exm:year 1829 ] .
exm:SchrHorn exm:name "Schreckhorn";
             exm:range exm:BerneseAlps;
             exm:height 4078;
             exm:location exl:Switzerland .
exm:GrGlck exm:name "Grossglockner";
           exm:range exm:HoheTauern;
           exm:height 3798;
           exm:location exl:Austria;
           exm:firstAscent [ exm:year 1800 ] .
```

```
exm:Nordend exm:name "Nordend";
            exm:range exm:PennineAlps;
            exm:height 4609;
            exm:location exl:Switzerland , exl:Italy;
            exm:firstAscent [ exm:year 1861 ] .
```

Wenn auf diesem Graph die SPARQL-Anfrage aus Beispiel 2.10 ausgeführt wird, erhält man das folgende Resultat.

?m	?n
http://www.example.org/mountains/MontBlanc	"Mont Blanc"
http://www.example.org/mountains/DufSptz	"Dufourspitze"
http://www.example.org/mountains/FstrHorn	"Finsteraarhorn"
http://www.example.org/mountains/SchrHorn	"Schreckhorn"
http://www.example.org/mountains/GrGlck	"Grossglockner"
http://www.example.org/mountains/Nordend	"Nordend"

□

In den folgenden Unterabschnitten wird die Syntax und Semantik von SPARQL beschrieben, wobei für Details auf die SPARQL-Spezifikation [PS06] verwiesen sei.

2.3.2 Grundlegendes zur Syntax

Eine SPARQL-Anfrage besteht aus

- einem Prolog,
- der Spezifikation einer Ergebnisform,
- einer optionalen Festlegung der untersuchten Datenmenge,
- einer, aus Graphmustern zusammengesetzten WHERE-Klausel und
- einer Menge von optionalen Klauseln zur Anpassung der Lösungen.

Der Prolog kann eine BASE-Klausel [PS06, 2.1.1 Syntax for IRI] zur Angabe einer Basis-URI für die Auflösung relativer URIs und eine Menge von PREFIX-Klauseln für Präfix-Spezifikationen zur Festlegung von URI-Namensräumen enthalten. Mögliche Ergebnisformen sind

SELECT Bei Verwendung einer SELECT-Klausel [PS06, 10.2 Selecting Variables] werden alle Lösungen einer Anfrage, also Mengen von den für die gefundenen Übereinstimmungen gebundenen Variablen, zurückgegeben. Die zurückzugebenden Variablen können durch Aufzählung hinter dem Schlüsselwort `SELECT` ausgewählt werden. Ein Sternchen hinter SELECT symbolisiert die Auswahl aller in der Anfrage erwähnten Variablen.

ASK Mit der ASK-Klausel [PS06, 10.5 Asking "yes or no"questions] kann gefragt werden, ob es für die Graphmuster überhaupt eine Übereinstimmung im untersuchten RDF-Graph gibt.

CONSTRUCT Die CONSTRUCT-Klausel [PS06, 10.3 Constructing an Output Graph] in Verbindung mit einem weiteren Graphmuster kann verwendet werden, um aus den Lösungen einen neuen RDF-Graph zu konstruieren. Bei der Konstruktion dient dieses Muster als Schablone, wobei die auftretenden Variablen durch die, entsprechend der Lösung an sie gebundenen URIs oder Literale ersetzt werden.

DESCRIBE Um einen RDF-Graph zu erhalten, welcher die gefundenen Lösungen beschreibt, kann die DESCRIBE-Klausel [PS06, 10.4 Descriptions of Resources] verwendet werden.

Die WHERE-Klausel bildet das Herz einer jeden Anfrage. Sie beschreibt das zusammengesetzte Graphmuster, welches eine beliebige Kombination der folgenden Graphmuster ist.

- *Einfaches Graphmuster* (engl.: basic graph pattern)
- *Graphmuster-Gruppe* (engl.: group graph pattern)
- *Optionales Graphmuster* (engl.: optional graph pattern)
- *Graphmuster-Vereinigung* (engl.: union graph pattern)
- *RDF-Graph-Klausel* (engl.: RDF dataset graph pattern)
- *Wertebeschränkungen* (engl.: value constraints)

Diese Graphmuster sind in der SPARQL-Spezifikation definiert und werden im Anschluss genauer beschrieben. Das zusammengesetzte Graphmuster einer SPARQL-Anfrage, welches mittels der WHERE-Klausel spezifiziert wird, beschreibt die Lösungen der Anfrage und wird demzufolge bei der Anfrageausführung für die Musterübereinstimmung verwendet.

Laut der SPARQL-Spezifikation werden Lösungen von Graphmustern durch sogenannte *Graphmusterlösungen* [PS06, 2.4 Pattern Solutions] repräsentiert. Das sind Ersetzungsfunktionen, welche jeder Variable ein *RDF-Terminal* (engl.: RDF term) [PS06, 2.2 Initial Definitions], also eine URI, ein Literal oder einen leeren Knoten, zuordnen. Im Rahmen dieser Arbeit wird ein Klasse von solchen Graphmusterlösungen durch eine Variablenbindung repräsentiert.

Definition 2.1 Sei S eine Graphmusterlösung [PS06, 2.4 Pattern Solutions], welche für ein bestimmtes Graphmuster P eine Musterübereinstimmung in einem bestimmten RDF-Graph G repräsentiert. Die Teilmenge von S, deren Definitionsbereich auf die in P erwähnten Anfrage-Variablen [PS06, 2.2 Initial Definitions] beschränkt ist, wird in Bezug auf P und G als *Variablenbindung* bezeichnet. □

Jede Variablenbindung repräsentiert demzufolge eine Teilmenge der Menge aller Graphmusterlösungen. In dieser Teilmenge sind alle Graphmusterlösungen enthalten, bei denen den Variablen, welche im Definitionsbereich der Variablenbindung enthalten sind, dieselben Werte zugeordnet werden wie durch die Variablenbindung spezifiziert. Demzufolge können die Variablenbindungen ebenfalls als Lösungen der Graphmuster gesehen werden.

2.3.3 Einfache Graphmuster

Einfache Graphmuster [PS06, 2.5 Basic Graph Patterns] bestehen aus einer Menge von *Tripelmustern* [PS06, 2.3 Triple Patterns]. Das sind Tripel, in denen Variablen an Stelle der RDF-Ausdrücke für Subjekt, Prädikat und Objekt stehen können. In SPARQL werden einfache Graphmuster in einer an Turtle angelehnten Darstellung notiert, wobei Variablen durch ein vorangestelltes Frage- oder Dollarzeichen gekennzeichnet sind. Eine Bindung aller Variablen an RDF-Terminale, also an URIs, Literale und leere Knoten ist eine Lösung für ein einfaches Graphmuster in Bezug auf einen RDF-Graphen, wenn alle enthaltenen Tripelmuster nach Ersetzung der Variablen mit dem Graph übereinstimmen, der Graph also entsprechende Tripel enthält. Die Beispiele 2.10 und 2.11 verdeutlichen diese Musterübereinstimmung.

Eine *Graphmuster-Gruppe* [PS06, 4.1 Group Graph Patterns] enthält eine Menge von Graphmustern. Eine Bindung aller Variablen an RDF-Terminale ist Lösung einer solchen Gruppe, wenn sie auch Lösung aller enthaltenen Graphmuster ist.

2.3.4 Optionale Informationen

Die *optionalen Graphmuster* [PS06, 5.4 Optional Matching – Formal Definition] dienen dem Abfragen von unvollständigem Wissen, da Daten für Semantic-Web-Anwendung unbekannt oder nicht verfügbar sein können. Hierfür wird in SPARQL einem Graphmuster das Schlüsselwort `OPTIONAL` vorangestellt, um es als optional zu kennzeichnen. Ein optionales Graphmuster (G, O) besteht aus zwei Graphmustern G und O, wobei eine Lösung von (G, O) entweder eine Lösung von O und G oder nur von G ist. Im zweiten Fall sind die Variablen, welche lediglich in O erwähnt werden, in der Lösung ungebunden [PS06, 4.2 Unbound variables].

Beispiel 2.12 *Die SPARQL-Anfrage*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n ?y
WHERE {
   ?m exm:name ?n .
   OPTIONAL {
      ?m exm:firstAscent ?a .
      ?a exm:year ?y . }
}
```

enthält ein optionales Graphmuster. Sie fragt nach dem Namen aller Berge und, sofern bekannt, nach dem Jahr der Erstbesteigung. Auf den in Beispiel 2.11 dargestellten RDF-Graph angewandt, liefert sie das folgende Resultat.

?n	?y
"Mont Blanc"	1786
"Dufourspitze"	1855
"Finsteraarhorn"	1829
"Schreckhorn"	
"Grossglockner"	1800
"Nordend"	1861

Weil der Graph keine Informationen über die Erstbesteigung des Schreckhorn enthält, ist in der entsprechenden Lösung die Variable `?y` *ungebunden.* □

2.3.5 Alternativen

Zur Beschreibung von Lösungsalternativen können *Graphmuster-Vereinigungen* [PS06, 6.2 Union Matching – Formal Definition] verwendet werden. In SPARQL werden hierfür einzelne Graphmuster durch das Schlüsselwort `UNION` verkettet. Jede Lösung einer Graphmuster-Vereinigung muss Lösung mindestens eines, der vereinigten Graphmuster sein.

Beispiel 2.13 *Die folgende SPARQL-Anfrage fragt nach den Namen aller Berge, welche sich entweder in den Berneser oder den Penniner Alpen befinden. Hierbei wird eine Graphmuster-Vereinigung benutzt.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n
WHERE {
   {
     ?m exm:name ?n ;
        exm:range exm:BerneseAlps .
   }
   UNION
   {
     ?m exm:name ?n
        exm:range exm:PennineAlps .
   }
}
```

Eine Anwendung auf den RDF-Graph aus Beispiel 2.11 liefert das folgende Resultat, welches den Grossglockner nicht beinhaltet, weil dieser in den Hohen Tauern liegt.

?n
"Mont Blanc"
"Dufourspitze"
"Finsteraarhorn"
"Schreckhorn"
"Nordend"

□

2.3.6 Wertebeschränkungen

Ein weiteres Graphmuster sind die Wertebeschränkungen [PS06, 3.3 Value Constraints – Definition]. Sie werden in SPARQL durch Boole'sche Ausdrücke in einer FILTER-Klausel beschrieben und dienen zur Einschränkung der möglichen Lösungen einer Anfrage, indem sie Bedingungen für die, an bestimmte Variablen gebundenen RDF-Terminale formulieren. Die Lösung einer Wertebeschränkung ist also eine Bindung aller Variablen an RDF-Terminale, für die der Ausdruck den Wahrheitswert $\top$ liefert.

Beispiel 2.14 *Die Frage nach den Namen aller Berge mit einer Höhe über 4500 Meter, kann in SPARQL durch die folgende Anfrage mit Wertebeschränkung spezifiziert werden.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n
WHERE {
   ?m exm:name ?n ;
      exm:height ?h .
   FILTER ( ?h > 4500 )
}
```

Auf den RDF-Graph aus Beispiel 2.11 angewandt, ergibt sich das folgende Resultat.

?n
"Mont Blanc"
"Dufourspitze"
"Nordend"

□

2.3.7 Datenmengen und benannte Graphen

Eine SPARQL-Anfrage wird auf einer *RDF-Datenmenge* [PS06, 7 RDF Dataset] ausgeführt, welche aus einem *Default-Graph* (engl.: default graph) und einer, möglicherweise leeren Menge von *benannten Graphen* (engl.: named graphs) besteht. Jeder benannte Graph ist durch eine URI identifiziert. Die Grundlage der Musterübereinstimmung bei einer Anfrageausführung ist zunächst immer der Default-Graph. Mittels *RDF-Graph-Klauseln* (engl.: RDF dataset graph pattern) [PS06, 8 Querying the Dataset] können jedoch explizit bestimmte benannte Graphen für die Musterübereinstimmung verwendet werden. In SPARQL werden solche RDF-Graph-Klauseln durch eine GRAPH-Klausel dargestellt. Sie besteht aus dem Schlüsselwort `GRAPH`, einer Graphspezifikation und einem Graphmuster, welches mit dem spezifizierten, benannten Graph übereinstimmen muss. Für die Graphspezifikation gibt es die folgenden Möglichkeiten.

- URI – Eine URI legt genau einen benannten Graph für die Musterübereinstimmung fest. Dabei handelt es sich um den durch die angegebene URI repräsentierten RDF-Graph. In diesem Fall ist eine Bindung aller Variablen an RDF-Terminale genau dann eine Lösung der RDF-Graph-Klausel, wenn sie Lösung des Graphmusters in der GRAPH-Klausel in Bezug auf den durch die URI repräsentierten RDF-Graph ist.

- leerer Knoten – Durch einen leeren Knoten werden alle benannten RDF-Graphen spezifiziert. Jede Lösung des Graphmusters in der GRAPH-Klausel in Bezug auf einen der benannten RDF-Graphen ist auch Lösung der RDF-Graph-Klausel.

- Variable – Dieser Fall entspricht dem obigen Fall der Spezifikation mittels leerem Knoten. Zusätzlich wird in jeder Lösung die angegebene Variable an die URI des benannten RDF-Graphen gebunden, welcher die Lösung liefert.

Über alle Graphmuster die sich innerhalb von RDF-Graph-Klauseln befinden kann gesagt werden, sie befinden sich im *Gültigkeitsbereich* (engl.: scope) dieser RDF-Graph-Klausel. Graphmuster innerhalb von verschachtelten RDF-Graph-Klauseln befinden sich im Gültigkeitsbereich der innersten, sie umgebenden RDF-Graph-Klausel. Demzufolge befindet sich die innere von zwei verschachtelten RDF-Graph-Klauseln im Gültigkeitsbereich der umgebenden.

Beispiel 2.15 *Die Informationen im RDF-Graph aus Beispiel 2.11 könnten auf verschiedene Graphen verteilt werden. Der folgende, in Turtle-Notation dargestellte Graph wird im Folgenden durch die URI* `http://www.example.org/mountains/FstrHorn.rdf` *repräsentiert und enthält Informationen über das Finsteraarhorn.*

```
@prefix exm: <http://www.example.org/mountains/> .

exm:FstrHorn exm:name "Finsteraarhorn";
             exm:range exm:BerneseAlps;
             exm:height 4274;
             exm:firstAscent [ exm:year 1829 ] .
```

Analog beschreibt der folgende, durch `http://www.example.org/mountains/GrGlck.rdf` *repräsentierte Graph den Grossglockner.*

```
@prefix exm: <http://www.example.org/mountains/> .

exm:GrGlck exm:name "Grossglockner";
           exm:range exm:HoheTauern;
           exm:height 3798;
           exm:firstAscent [ exm:year 1800 ] .
```

Die folgende SPARQL-Anfrage enthält eine Graphmuster-Gruppe, welche wiederum eine RDF-Graph-Klausel der Form `GRAPH <uri> {...}` *und ein einfaches Graphmuster enthält. Das Graphmuster innerhalb der GRAPH-Klausel ist ebenfalls ein einfaches Graphmuster. Es befindet sich im Gültigkeitsbereich der GRAPH-Klausel.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n ?l
WHERE {
   GRAPH <http://www.example.org/mountains/FstrHorn.rdf> {
      ?m exm:name ?n .
   }
   ?m exm:location ?l .
}
```

Auf eine RDF-Datenmenge bestehend aus dem RDF-Graph aus Beispiel 2.11 als Default-Graph und den beiden benannten Graphen `http://www.example.org/mountains/FstrHorn.rdf` *und* `http://www.example.org/mountains/GrGlck.rdf` *angewandt, ergibt die Anfrage das folgende Resultat.*

?n	?l
"Finsteraarhorn"	http://www.example.org/locations/Switzerland

□

Beispiel 2.16 *Wenn die SPARQL-Anfrage*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?g ?n
WHERE {
   GRAPH ?g { ?m exm:name ?n . }
}
```

auf die RDF-Datenmenge aus Beispiel 2.15 angewandt wird, ergibt sich das folgende Resultat.

?g	?n
http://www.example.org/mountains/FstrHorn.rdf	"Finsteraarhorn"
http://www.example.org/mountains/GrGlck.rdf	"Grossglockner"

□

Die für die Ausführung einer SPARQL-Anfrage zu untersuchende RDF-Datenmenge kann direkt in der Anfrage spezifiziert werden. Hierzu dienen die Klauseln FROM und FROM NAMED [PS06, 9 Specifying RDF Datasets]. Mit ersterer kann ein RDF-Graph, repräsentiert durch eine URI, für den Default-Graph angegeben werden. Bei mehreren FROM-Klauseln wird der Default-Graph durch eine Verschmelzung der spezifizierten Graphen gebildet. Mit zweiterer kann jeweils ein benannter Graph, repräsentiert durch eine URI, festgelegt werden.

2.3.8 Anpassung der Lösungsmenge

Wie oben erwähnt können der WHERE-Klausel Angaben zur Anpassung der Lösungsmenge, also der Menge aller Lösungen des Graphmusters der Anfrage, folgen. Hierbei handelt es sich um die folgenden Klauseln, welche einzeln oder in Kombination angegeben werden können.

DISTINCT Mit dem Schlüsselwort `DISTINCT` [PS06, 10.1.2 DISTINCT] kann die Disjunktheit aller Lösungen gefordert werden. Die Verwendung von `DISTINCT` ist nur bei SELECT-Anfragen möglich.

ORDER BY Zur Sortierung der Lösungsmenge eine Anfrage wird in SPARQL das Schlüsselwort `ORDER BY` [PS06, 10.1.3 ORDER BY] mit einer darauf folgendenden Liste von *Ordnungsbedingungen* (engl.: ordering conditions) eingesetzt. Diese Ordnungsbedingungen können Variablen oder Funktionsaufrufe sein, welche für eine Sortierung der Lösungen sorgen. Im Normalfall werden die Lösungen aufsteigend sortiert, wobei die Sortierrichtung mit den Schlüsselwörtern ASC und DESC explizit angegeben werden kann. Bei mehreren Ordnungsbedingungen werden diese in der angegebenen Reihenfolge ausgeführt.

LIMIT Mit dem Schlüsselwort `LIMIT` [PS06, 10.1.4 LIMIT] und einer darauf folgenden Zahl kann die Kardinalität der Lösungsmenge auf den angegebenen Wert eingeschränkt werden.

OFFSET Dem Schlüsselwort `OFFSET` [PS06, 10.1.5 OFFSET] folgt ebenfalls eine Zahl. Deren Wert bestimmt den Index der ersten für das Resultat relevanten Lösung innerhalb der Lösungsmenge. Alle Lösungen vor der so spezifizierten werden ignoriert.

Auch wenn die SPARQL-Spezifikation LIMIT- und OFFSET-Klauseln ohne eine Angabe zur Sortierung der Lösungen nicht ausschließt sei darauf hingewiesen, dass diese in den meisten Fällen nur in Kombination sinnvoll sind.

Beispiel 2.17 *Die folgende SPARQL-Anfrage fragt nach dem Namen des zweithöchsten Berges. Hierfür werden Angaben zur Anpassung der Lösungsmenge benutzt.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n
WHERE {
   ?m exm:name ?n ; exm:height ?h .
}
ORDER BY ?h
OFFSET 2
LIMIT 1
```

Eine Anwendung auf den RDF-Graph aus Beispiel 2.11 liefert das folgende Resultat.

?n
"Dufourspitze"

□

2.4 Bearbeitung von Anfragen

Die in dieser Arbeit spezifizierten Begriffe und Modelle und die entwickelten Konzepte dienen der Bearbeitung von SPARQL-Anfragen. Um sie in diesem Rahmen einordnen zu können, gibt dieser Abschnitt einen Einblick in die Anfragebearbeitung.

Es gibt noch nicht sehr viele Publikationen, welche sich speziell mit der Bearbeitung von SPARQL-Anfragen beschäftigen. Jedoch weist die Bearbeitung von Datenbankanfragen viele Parallelen zur SPARQL-Bearbeitung auf. Deshalb lohnt es, sich mit dieser zu beschäftigen. Der erste Teil dieses Abschnitts beschreibt deshalb das verbreitete Vorgehen bei der Bearbeitung von Anfragen in Datenbank-Managementsystemen (DBMS) und stellt die wichtigsten Konzepte und Begriffe vor. Der zweite Teil verweist auf aktuelle Arbeiten, welche sich in unterschiedlichster Weise mit der Bearbeitung von Anfragen an RDF-Datenmengen und SPARQL beschäftigen.

2.4.1 Anfragebearbeitung in DBMS

Die meisten Sprachen für Datenbankenanfragen haben einen deklarativen Charakter. Die Benutzer beschreiben mittels der Anfrage die Daten, welche ihnen ein DBMS aus einer Datenbank liefern soll. Demzufolge wird kein Berechnungsweg zum Ermitteln der Resultate vorgegeben und das DBMS muss für jede Anfrage eine *Ausführungsstrategie* (engl.: evaluation strategy) bestimmen. Diese muss möglichst optimal sein, um die Zeit und Ressourcen für deren Ausführung in einem angemessenen Rahmen zu halten. Diese Voraussetzungen und Anforderungen treffen auch auf SPARQL und ein entsprechendes Anfragebearbeitungssystem zu.

Nach Garcia-Molina et al. [GMWU02] wird die *Anfragebearbeitung* (engl.: query processing) in einem DBMS in der Regel grob in die zwei Phasen *Anfragekompilierung* (engl.: query compilation) und *Anfrageausführung* (engl.: query execution) unterteilt (siehe Abbildung 2.3 auf Seite 16).

Während der Anfragekompilierung wird aus der zu bearbeitenden Anfrage ein *Anfrage-Ausführungsplan* (engl.: query execution plan – QEP) erzeugt, welcher eine möglichst optimale Ausführungsstrategie repräsentiert und die Funktionalität zur Ermittlung der angefragten Daten beschreibt. Dieser Plan wird bei der Anfrageausführung abgearbeitet und somit die Ergebnismenge bestimmt.

Es sei darauf hingewiesen, dass die Anfrageausführung in der Literatur auch oft als *Anfrageauswertung* (engl.: query evaluation) bezeichnet wird. Ähnliches gilt für die Anfragekompilierung. Da diese aus mehreren Schritten besteht, wird sie bereits in der groben Einteilung häufig durch zwei oder mehr Unterphasen repräsentiert. In [SKS97] und [Sch04] sind das beispielsweise die *Anfrageübersetzung* (engl.: query translation) und die *Anfrageoptimierung* (engl.: query optimization). In [GMWU02] sind es die Schritte *Parsen*, *Logische Planerzeugung* und *Physische Planerzeugung* und in [HFLP89] *Parsen*, *Anfrageumstrukturierung* (engl.: query rewrite), *Planoptimierung* (engl.: plan optimization) und *Planverfeinerung* (engl.: plan refinement). In allen Fällen sind die zu Grunde

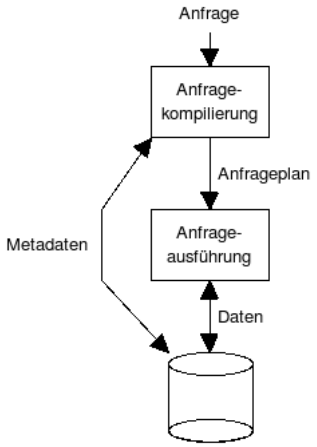


Abbildung 2.3: Hauptbestandteile der Anfragebearbeitungseinheit eines DBMS nach [GMWU02]

liegenden Konzepte und durchzuführenden Teilschritte zur Anfragekompilierung jedoch dieselben. Auf sie wird im Folgenden näher eingegangen.

Der erste Schritt der Anfragekompilierung ist die Anfrageübersetzung und -vereinfachung. Deren Ergebnis ist eine für die Optimierung geeignete interne Darstellung der Anfrage, häufig in Form eines Operatorbaums. Hierfür wird die als Zeichenkette gegebene Anfrage zuerst von einem Parser lexikalisch, syntaktisch und semantisch analysiert und dann mittels Transformationsregeln in einen Operatorbaum umgewandelt. In vielen RDBMS entspricht dieser Operatorbaum beispielsweise einem Term der relationalen Algebra, welche unter anderem in [GMWU02, Seite 189ff.] ausführlich vorgestellt wird. Im nächsten Teilschritt werden eventuell in der Anfrage verwendete Sichten aufgelöst, was eine Erweiterung des Operatorbaums um Teilbäume, welche die Sichten repräsentieren, zur Folge hat. Daraufhin wird die Anfrage in eine Normalform überführt, wobei bestimmte Äquivalenzbeziehungen zwischen den Operatoren ausgenutzt werden. Diese standardisierte Darstellung der Anfrage wird letztendlich noch von Redundanzen und Inkonsistenzen befreit.

Die logische Optimierung ist der nächste Schritt der Anfragekompilierung. Sie wird auch als regelbasierte oder algebraische Optimierung bezeichnet und liefert einen, durch Anwendung von Transformationsregeln, umgeformten Operatorbaum. Die Anwendbarkeit dieser Umformungsregeln ist durch Äquivalenzbeziehungen zwischen den Operatoren gerechtfertigt. Das Ziel der Anfrageumschreibungen ist durch verschiedene Heuristiken festgelegt, welche in den meisten Fällen eine Verkleinerung der Zwischenergebnisse und damit eine schnellere Anfrageausführung versprechen. In [YM98] werden die bekanntesten Heuristiken für die relationale Algebra vorgestellt. So sollen

- Selektionen und Projektionen so früh wie möglich ausgeführt,
- kartesische Produkte möglichst durch Verbundoperationen ersetzt und
- die einschränkensten Verbundoperationen zuerst ausgeführt

werden. Für diese Form der Optimierung wird kein Wissen über das interne Schema der Datenbank und keine Informationen über die Daten benötigt.

Der abschließende und aufwendigste Schritt der Anfragekompilierung ist die physische Optimierung mit kostenbasierter Auswahl, die den auszuführenden Plan liefert. Während der physischen Optimierung, welche auch oft als interne Optimierung bezeichnet wird, werden verschiedene, alternative Anfrage-Ausführungspläne erzeugt. Diese Pläne basieren auf dem Operatorbaum, dessen abstrakte Operatoren in physische Operatoren umgewandelt werden. Bei der Umwandlung

werden für die einzelnen Operatoren Algorithmen zu ihrer Implementierung festgelegt. Für die meisten abstrakten Operationen stehen hierbei verschiedene, alternative Algorithmen [GMWU02, Seite 722ff.] mit unterschiedlichen Eigenschaften zur Auswahl. Die Ausführungspläne beinhalten zusätzlich Operatoren für den Zugriff auf die Datenstrukturen, beispielsweise auf die Relationen. Auch hierfür gibt es alternative Möglichkeiten, welche sich beispielsweise durch die Verwendung von Indexen ergeben. Demzufolge muss bei der physischen Optimierung das Datenbankschema bekannt sein.

Eine weitere Möglichkeit der physischen Optimierung ist die *datenstrombasierte Verarbeitung* (engl.: pipelining) [SKS97, Seite 413ff.] bei der physische Operatoren während der Ausführung verkettet werden, so dass die einzelnen Ergebnistupel direkt vom jeweils folgenden Operator weiterverarbeitet werden. Hierbei entstehen keine vollständigen Zwischenergebnisse, welche Speicherplatz benötigen, und die Ergebnisrückgabe wird beschleunigt, da Operatoren nicht mehr auf die Fertigstellung des vollständigen Ergebnisses ihrer Vorgänger warten müssen. Eine verbreitete Möglichkeit zur Implementierung der datenstrombasierten Verarbeitung bieten Iteratoren [GMWU02, Seite 720ff.]. Es sei jedoch darauf hingewiesen, dass nicht alle Algorithmen, welche Operatoren implementieren, für die datenstrombasierte Verarbeitung geeignet sind.

Von den verschiedenen Ausführungsplänen muss ein möglichst optimaler ausgewählt werden. Dazu werden die Kosten, in Bezug auf Rechen- und Datenzugriffszeit, Speicherplatz und Kommunikationszeit bei verteilter Anfrageausführung, für jeden Plan nach einem Kostenmodell bestimmt und miteinander verglichen. Nach Saake et al. [GS05] sind die wichtigsten Komponenten der Kostenmodelle Kostenfunktionen zur Abschätzung der Kosten für Operationen [SKS97, Seite 386ff.], Statistiken über die Größe, Wertebereiche und Verteilungen von Daten [SKS97, Seite 384ff.] und Formeln zur Bestimmung der Größe von Zwischenergebnissen auf Basis der Statistiken [GS05, Seite 442ff].

Aufgrund der vielfältigen Möglichkeiten der physischen Optimierung könnten sehr viele Ausführungspläne generiert werden, wodurch der Suchraum für die kostenbasierte Auswahl sehr groß wird. Um zu vermeiden, dass die Ausführungszeit der Anfrageoptimierung in die Größenordnung der Zeit für die Anfrageausführung geht, werden physische Optimierung und kostenbasierte Auswahl meistens miteinander kombiniert. In den meisten Fällen wird dabei der Suchraum dynamisch eingeschränkt oder über iterative Näherungsverfahrung ein möglichst optimaler Plan bestimmt. Die bekanntesten Verfahren werden in [GMWU02, Seite 842ff.] vorgestellt.

Wie bereits erwähnt wird in der zweiten Phase der Anfragebearbeitung der festgelegte Plan ausgeführt. Dieser Plan kann als Programm zur Bestimmung des Anfrageergebnisses aufgefasst werden, welches in den meisten Fällen von einem Interpreter abzuarbeiten ist. Hierbei wird die Ergebnismenge erzeugt und bereitgestellt. Der Plan von wiederholt auszuführenden Anfragen wird bei einigen Systemen aufbewahrt oder sogar in ausführbaren Code umgewandelt.

2.4.2 Forschungsstand der Bearbeitung von SPARQL-Anfragen

Die SPARQL-Spezifikation ist eine der derzeit aktuellsten Entwicklungen im Rahmen der Semantic-Web-Aktivitäten des W3C. Demzufolge existieren momentan nur wenige Publikationen die sich mit der Bearbeitung von SPARQL-Anfragen beschäftigen und es konnte keine Veröffentlichung gefunden werden, welche sich umfassend mit diesem Thema auseinandersetzt. Die in den Texten vorgestellten Forschungsergebnisse betreffen immer nur einzelne Aspekte der Anfragebearbeitung und werden im Folgenden vorgestellt.

Die derzeit wahrscheinlich umfassenste Arbeit [PAG06] beschäftigt sich mit der Semantik von Graphmustern in SPARQL und dem Vorgehen und Komplexität, Lösungen dafür zu finden. Zunächst werden Graphmuster formal, in Form einer Algebra definiert, die Semantik dieser Darstellung spezifiziert und Äquivalenzbeziehungen algebraischer Ausdrücke vorgestellt und bewiesen. Auf Basis der Algebra wird dann ein Auswertungsproblem benannt und analysiert, dessen Lösung zu einer Menge von Musterübereinstimmungen der Graphmuster bezüglich einer RDF-Datenmenge führt. Diese Strategie zur Ausführung von SPARQL-Anfragen wird als *Kompositionsansatz* bezeichnet, da hierbei zur Bestimmung der Lösungen eines Ausdrucks die Lösungen aller Teilausdrücke bestimmt und kombiniert werden müssen. Demzufolge wird die Gesamtlösung durch

ein *Bottom-Up-Vorgehen* ermittelt. Als weitere Lösungsstrategie stellen Gutierrez et al. [PAG06] zusätzlich einen prozeduralen Ansatz vor. Dieser beruht auf einer funktionalen Definition der Semantik von Graphmustern und bestimmt die Lösungsmenge durch eine Tiefensuche, bei der die Auswertung von Teilen des Graphmusters zu einzelnen Variablenbindungen führt, die dann für die Auswertung weiterer Teile festgelegt sind. Ein solches Vorgehen wird beispielsweise in ARQ[3] [Sea06], dem Anfragebearbeitungssystem des Semantic-Web-Frameworks Jena[4], und der SPARQL-Komponente sparql-p [Her05] der RDFLib-Bibliothek[5] verfolgt.

Bei beiden Ausführungsstrategien müssen Lösungen für die, in der Anfrage enthaltenen Graphmuster gefunden werden. Ein wichtiger Punkt hierbei ist die Bestimmung von Musterübereinstimmungen zwischen Tripelmustern und RDF-Graphen. Diese könnten unter anderem durch eine Methode gefunden werden, die von den in [Car02] vorgestellten Algorithmen zum Testen zweier RDF-Graphen auf Äquivalenz abgeleitet ist.

Eine dritte, mögliche Strategie zur Ausführung von SPARQL-Anfragen, bei der nicht explizit eine Musterübereinstimmung ausgeführt werden muss, ist die Bestimmung von Ergebnissen durch logisches Schließen. Ein solches Vorgehen wird in [dBFT05] diskutiert. Hier wird die Anfrageauswertung auf ein Problem der *Beschreibungslogik* (engl.: description logics) übertragen und kann dann mittels eines entsprechenden Lösungsverfahrens ausgeführt werden.

Im Gegensatz zu anderen vorgeschlagenen Anfragesprachen für RDF-Datenmengen [ACK+02, S06] beschränkt sich die SPARQL-Spezifikation auf die RDF-Semantik, wodurch zunächst keine Lösungen gefunden werden können, die unter Verwendung von Schemadaten und Ontologiedefinitionen implizit in der Datenmenge enthalten sind. Es gibt jedoch mehrere Möglichkeiten, diese Einschränkung zu überwinden. In [GHM04] wird eine, auf Logik basierende Strategie zur Anfrageausführung beschrieben, bei der die verwendete RDF-Datenmenge mittels Schlussregeln bezüglich einer RDF-Schema-Ontologie während der Anfrageausführung vervollständigt wird. Es ist natürlich auch möglich die Datenmenge mit Hilfe eines Reasoners bereits vor der Ausführung zu vervollständigen. Das bietet sich insbesondere für den Fall an, dass die Daten in einer Datenbanken gespeichert werden. Dieser Ansatz wird in [PH04] vorgestellt und ist beispielsweise in Jena in Form eines der möglichen Daten-Modelle implementiert. Eine andere Methode ist die in [SKT05] beschriebene *semantische Anfrageoptimierung* (engl.: semantic query optimization – SQO), die auf der Idee einer Erweiterung der Anfrage bezüglich der RDF-Schema-Semantik basiert. Hierbei wird eine Anfrage zunächst durch eine Algebra beschrieben. In dieser Form wird die Anfrage dann nach festgelegten Regeln für das Schema erweitert. Der erhaltene algebraische Ausdruck kann dann durch ein entsprechendes Anfragebearbeitungssystem ausgeführt oder in die ursprüngliche Anfragesprache zurücktransformiert werden.

Für jede der erwähnten Lösungsstrategien ist die interne Repräsentation der Anfrage eine wichtige Grundlage der Bearbeitung. In [FHVB04] wird beispielsweise die Algebra RAL zur Repräsentation von Anfragen an RDF-Datenmengen spezifiziert, die sich an der relationalen Algebra orientiert und deren Operatoren auf Mengen von Knoten eines RDF-Graphen definiert sind. Das Dokument lässt jedoch Transformationsregeln für die Übersetzung einer konkreten RDF-Anfragesprache in die Algebra vermissen. Diese lassen sich aber sicher relativ einfach definieren. Erwähnenswert ist [FHVB04] ausserdem, weil hier Umformungsregeln für RAL-Ausdrücke angegeben und diese für die Anwendung von Heuristiken eingesetzt werden. Die angegebenen Heuristiken sind an die, bei der Anfrageoptimierung in DBMS angewandten, angelehnt. Zur Ausführung der restrukturierten Anfrage wird hier die Tiefensuche im Operatorbaum als Strategie zur Bestimmung von Lösungen vorgeschlagen und kurz skizziert.

LAGAR ist eine andere Algebra, die in [CGK05] vorgestellt wird und ebenfalls an die relationale Algebra angelehnt ist. Die Operatoren von LAGAR sind, im Gegensatz zu den RAL-Operatoren, auf Mengen von RDF-Graphen definiert und die zu Grunde liegenden Graphmuster werden be-

[3] `http://jena.sourceforge.net/ARQ/`
[4] `http://jena.sourceforge.net/`
[5] `http://rdflib.net/`

züglich der RDF-Schema-Semantik interpretiert, wodurch bei einer Verwendung der Algebra für SPARQL die erwähnte Einschränkung auf die RDF-Semantik aufgehoben werden könnte.

Weitere wichtige Punkte einer Anfrageoptimierung sind für die Erzeugung von Ausführungsplänen verwendbare Index-Strukturen und Kostenmodelle für die Auswahl des optimalsten Plans. Eine mögliche Index-Struktur für RDF-Datenmengen auf Basis von B^+-Bäumen wird in [HD05] beschrieben. Diese könnte bei der Ausführung von SPARQL-Anfragen verwendet werden. In [RvGB03] wird das auf Hash-Verfahren basierte Speichermodell RDFStore für RDF-Daten vorgestellt und in diesem Zusammenhang Index-Strukturen auf Basis von mehrdimensionalen, schwach besetzten Matrizen spezifiziert. Stuckenschmidt et al. [SVBH04] beschäftigen sich mit der verteilten Bearbeitung von Anfragen an RDF-Datenmengen und stellen in diesem Rahmen ebenfalls Index-Strukturen und ein Kostenmodell vor, welche sich auch auf andere Anwendungsfälle übertragen lassen.

Zum Abschluss dieses Abschnitts sei noch das verhältnismässig gut erforschte Teilgebiet der Bearbeitung von Anfragen an RDF-Datenmengen, welche in herkömmlichen Datenbanken gehalten werden, erwähnt. Viele Semantic-Web-Anwendungen müssen große Mengen an RDF-Daten persistent speichern und oft darauf zugreifen können. Hierfür bietet sich der Einsatz von Datenbank-Managementsystemen an, deren Fähigkeiten, wie Transaktionen, Zugriffsrechte-Verwaltung, Back-up-Möglichkeiten, hohe Skalierbarkeit und viele mehr, für die Anwendungen nützlich sind. Demzufolge können bei vielen, auf RDF basierenden Frameworks, Werkzeugsammlungen und Bibliotheken die Datenmengen in Datenbanken gehalten werden. Die wichtigsten davon werden in [BG03] vorgestellt.

Zur konkreten Speicherung von RDF-Daten in einer Datenbank exisiteren verschiedene Ansätze und demzufolge gibt es viele Publikationen, in denen die unterschiedlichsten Datenbank-Schemata vorgestellt werden. Ein guter Überblick ist in [BG03] und [TCK05] zu finden, wobei [TCK05] einen Vergleich in Form eines Benchmarks enthält.

Ein weiterer Vorteil von Datenbanken als Speicher für RDF-Datenmengen sind die Anfragebearbeitungskomponenten in den Datenbank-Managementsystemen mit einer, seit vielen Jahren untersuchten und stetig verbesserten Anfrageoptimierung. Diese Optimierung kann auch bei Anfragen an persistent gespeicherte RDF-Datemengen ausgenutzt werden, indem die Datenbank direkt über eine erweiterte Variante der Datenbank-Anfragesprache SQL angefragt wird [CDES05] oder die Anfragebearbeitungskomponente der RDF basierten Frameworks möglichst große Teile der Anfrageausführung an das Datenbanksystem überträgt [ACK+01, WSKR03, HG03]. Seit SPARQL begonnen hat, sich als die Anfragespache für RDF-Datenmengen durchzusetzen, werden auch spezielle Möglichkeiten für die SPARQL-Bearbeitung in Zusammenhang mit persistent gespeicherten RDF-Daten entwickelt. So wird in [Cyg05] eine relationale Algebra für SPARQL definiert und eine Abbildung dieser auf SQL-Anfragen angegeben. Diese Arbeit bildet die Grundlage des Anfragebearbeitungssystems sparql2sql[6], welches aus einer SPARQL-Anfrage eine SQL-Anfrage bezüglich des in Jena verwendeten Datenbank-Schemas [WSKR03] erzeugt. Ein analoger Ansatz wird in dem RDF-Speicher-System 3store[7] verfolgt. In [Har05] wird zunächst das in 3store verwendete Datenbank-Schema vorgestellt und dann eine Transformation von SPARQL, über eine Algebra nach SQL für dieses Schema angegeben.

Die in diesem Abschnitt erwähnten Konzepte und Ideen können in die Entwicklung einer Bearbeitungskomponente für SPARQL-Anfragen einfließen. Das zu Beginn erwähnte Fehlen einer umfassenden Thematisierung der Anfragebearbeitung für SPARQL wird in dieser Arbeit ausgeglichen, indem der Abschnitt 3.3 (ab Seite 33) einen zusammenfassenden Überblick über die möglichen Schritte der Anfragekompilierung gibt. Die Grundlage der Umwandlung, im Rahmen dieser Arbeit, ist ein Modell, welches SPARQL-Anfragen repräsentiert und im folgenden Kapitel vorgestellt wird.

[6] `http://jena.sourceforge.net/sparql2sql/`
[7] `http://threestore.sourceforge.net/`

Kapitel 3

Das SPARQL Query Graph Model

Wie in Abschnitt 2.4 (ab Seite 15) erwähnt, bildet eine interne Repräsentation der Anfrage die Grundlage einer maschinellen Anfrageoptimierung und Planerzeugung. Im Rahmen des Starburst-Projekts [HCL+90] wurde das Query Graph Model (QGM) für SQL-Anfragen eingeführt. In Anlehnung an dieses Modell wird in diesem Kapitel das SPARQL Query Graph Model (SQGM) zur Repräsentation von SPARQL-Anfragen vorgestellt. Dieses Modell repräsentiert SPARQL-Anfragen in einer einfachen aber vollständigen Form, welche die Möglichkeit für eine umfangreiche Anfrageanalyse, Transformation und Optimierung bietet. Demzufolge ist es auch die Basis der weiteren Diskussion innerhalb dieser Arbeit.

Die Hauptidee des SQGM ist die Repräsentation der Teile einer Anfrage durch verschiedene Operatoren. Einzelne Operatoren sind durch Datenflüsse miteinander verbunden. Dieses Modell bietet die Möglichkeit zur Darstellung in Form eines gerichteten Graphen.

Dieses Kapitel ist in die folgenden drei Abschnitte unterteilt. Zu Beginn wird das SQGM formell definiert. Hierbei werden hauptsächlich die unterschiedlichen Operatoren mit ihren Eigenschaften spezifiziert und in Beziehung zu den repräsentierten Anfrageteilen gesetzt. Im darauf folgenden, zweiten Abschnitt wird die erwähnte Möglichkeit zur Darstellung eines SQGM als Graph spezifiziert und durch einige Beispiele bereichert. Der abschließende dritte Abschnitt beschreibt ein mögliches Vorgehen bei der Bearbeitung von SPARQL-Anfragen auf Basis des SQGM.

3.1 Formale Definition des SQGM

3.1.1 Grundlegende Definitionen

Die Bestandteile eines SQGM sind Operatoren und Datenflüssen zwischen den Operatoren. Während die Operatoren bestimmte Operationen über Eingangsdaten ausführen und dabei Ausgangsdaten erzeugen, werden diese Daten mit Hilfe der Datenflüsse zwischen den Operatoren übermittelt. Bei diesen Daten kann es sich um RDF-Graphen oder Mengen Variablenbindungen handeln. Die RDF-Graphen, welche wie in Abschnitt 2.2 (ab Seite 4) bereits erwähnt Tripelmengen darstellen, repräsentieren Aussagen über Ressourcen. Die Variablenbindungen sind Lösungen von Graphmustern. Wie in Abschnitt 2.3.2 (ab Seite 9) erwähnt sind diese Lösungen Ersetzungsfunktionen, welche jeder Variable ein RDF-Terminal, also eine URI, ein Literal oder einen leeren Knoten, zuordnen.

Die folgenden Definitionen führen die neuen Begriffe für die weitere Verwendung im Rahmen dieser Arbeit formal ein.

Definition 3.1 Ein *Operator* führt auf einer bestimmten Menge von Eingangsdaten eine bestimmte Operation aus und erzeugt dabei Ausgangsdaten. Die konkrete Operation ist vom Typ des Operators abhängig. Die Eingangs- und Ausgangsdaten können jeweils

- ein RDF-Graph [KC04, 6.2 RDF Graph],
- eine Menge von Variablenbindungen (siehe Definition 2.1 auf Seite 10) oder
- ein Wahrheitswert T oder F

sein. Die Menge OP enthält alle Operatoren eines SQGM. □

Wie in der Definition erwähnt existieren verschiedene Typen von Operatoren, welche unterschiedliche Operationen ausführen. Die einzelnen Operatortypen werden im Folgenden spezifiziert. Aus jedem Operator könnte bei der Plangenerierung ein Teil des Ausführungsplans erzeugt werden.

Jeder Operator kann verwendet werden, um einen bestimmten Teil einer SPARQL-Anfrage zu modellieren. Dabei kommen für bestimmte Teile nur Operatoren eines bestimmten Typs in Frage. Wie aus einer SPARQL-Anfrage ein SQGM erzeugt wird ist in Abschnitt 4.1 (ab Seite 38) beschrieben.

Wie bereits erwähnt sind die einzelnen Operatoren durch Datenflüsse miteinander verbunden.

Definition 3.2 Ein *Datenfluss* d verbindet zwei Operatoren o_1 und o_2 miteinander indem es die Ausgangsdaten des Operators o_1 dem Operator o_2 als Eingangsdaten zur Verfügung stellt. In diesem Zusammenhang wird o_1 als *bereitstellender Operator* und o_2 als *benutzender Operator* bezeichnet. Die Ausgangsdaten von o_1 werden bezüglich d als *übermittelte Daten* bezeichnet. Die Menge DF enthält alle Datenflüsse eines SQGM. □

Die Datenflüsse, welche mit einem Operator verknüpft sind, werden durch die im Folgenden definierten Eigenschaften repräsentiert.

Definition 3.3 Alle Operatoren haben die folgenden beiden Eigenschaften.

- *input* – Die Eigenschaft *input* repräsentiert die Datenflüsse, welche die Eingabedaten des entsprechenden Operators bereitstellen. Diese Datenflüsse werden als *Eingangsdatenflüsse* des Operators bezeichnet.
- *output* – Die Eigenschaft *output* repräsentiert die Datenflüsse, welche die Ausgabedaten des entsprechenden Operators anderen Operatoren als Eingabedaten zur Verfügung stellen. Diese Datenflüsse werden als *Ausgangsdatenflüsse* des Operators bezeichnet. □

Der Zugriff auf den Wert einer Eigenschaft von Objekten wird im Folgenden durch die Verknüpfung von Objektname und Eigenschaftsname mittels eines Punktes notiert. Beispielsweise repräsentiert die Notation $o.\text{output}$ den Wert der Eigenschaft *output* des Operators o, also die Ausgangsdatenflüsse von o. Wenn eine Eigenschaft ein Objekt repräsentiert für welches wiederum Eigenschaften definiert sind, so kann über eine weitere Punkt-Verknüpfung auf deren Werte zugegriffen werden.

Auf die in der Definition der Datenflüsse (Definition 3.2) erwähnten bereitstellenden und benutzenden Operatoren eines Datenflusses kann mit Hilfe der im Folgenden definierten Abbildungen zugegriffen werden.

Definition 3.4 Die Abbildung $\mathsf{UsingOp} : DF \rightarrow OP$ weist jedem Datenfluss den ihn benutzenden Operator zu, wobei

$$\mathsf{UsingOp}(d) := o \Longleftrightarrow d \in o.\text{input}$$

gilt. □

Definition 3.5 Die Abbildung $\mathsf{ProvOp} : DF \rightarrow OP$ weist jedem Datenfluss den ihn bereitstellenden Operator zu, wobei

$$\mathsf{ProvOp}(d) := o \Longleftrightarrow d \in o.\text{output}$$

gilt. □

3.1.2 Klassen von Operatoren

Die Menge der SQGM-Operatoren kann hauptsächlich in die zwei im Folgenden definierten Klassen eingeteilt werden, deren Unterscheidungsmerkmal die Art der bereitgestellten Daten ist.

Definition 3.6 Ein *graph-bereitstellender Operator* (engl.: graph providing operator) ist ein Operator, dessen Operation einen RDF-Graph [KC04, 6.2 RDF Graph] erzeugt. Die Menge $GPOP$ enthält alle graph-bereitstellenden Operatoren eines SQGM, wobei $GPOP \subset OP$ gilt. □

Definition 3.7 Ein *variablen-bereitstellender Operator* (engl.: variable providing operator) ist ein Operator, dessen Operation eine Menge von Variablenbindungen (siehe Definition 2.1 auf Seite 10) erzeugt. Die Menge $VPOP$ enthält alle variablen-bereitstellenden Operatoren eines SQGM, wobei $VPOP \subset OP$ gilt. Alle variablen-bereitstellenden Operatoren haben die folgende Eigenschaft.

- *provVars* – eine Menge von Anfrage-Variablen [PS06, 2.2 Initial Definitions], die dem Definitionsbereich der jeweils erzeugten Variablenbindungen entspricht □

Die Mengen von graph-bereitstellenden und variablen-bereitstellenden Operatoren eines SQGM sind disjunkt.

$$GPOP \cap VPOP = \emptyset$$

Entsprechend den beiden Hauptoperatortypen werden zwei Arten von Datenflüssen unterschieden.

Definition 3.8 Ein *Graphdatenfluss* (engl.: graph data flow) ist ein Datenfluss, der von einem graph-bereitstellenden Operator bereitgestellt wird und dessen übermittelte Daten demzufolge ein RDF-Graph sind. Die Menge GF enthält alle Graphdatenflüsse eines SQGM, wobei $GF \subset DF$ gilt. □

Definition 3.9 Ein *Variablendatenfluss* (engl.: variable data flow) ist ein Datenfluss, der von einem variablen-bereitstellenden Operator bereitgestellt wird und dessen übermittelte Daten demzufolge Mengen von Variablenbindungen sind. Die Menge VF enthält alle Variablendatenflüsse eines SQGM, wobei $VF \subset DF$ gilt. Die Menge der übergebenen Variablenbindungen ist durch die Eigenschaft

- *vars* – eine Menge von Anfrage-Variablen [PS06, 2.2 Initial Definitions], welche im Definitionsbereich der vom bereitstellenden Operator erzeugten Variablenbindungen enthalten sind

definiert, wobei

$$\forall\, d \in VF : d.\text{vars} \subseteq \mathsf{ProvOp}(d).\text{provVars}$$

gelten muss. □

Die Einschränkung der Eigenschaft *vars* gibt einerseits an, dass die mittels eines Variablendatenflusses übergebenen Variablenbindungen nur die vom bereitstellenden Operator bereitgestellten Variablen beinhalten können. Andererseits müssen aber nicht alle bereitgestellten Variablen übergeben werden. Das wäre beispielsweise der Fall, wenn eine Variable nur innerhalb eines Graphmusters g_{inner} aus einer Graphmuster-Gruppe g_{Gruppe} erwähnt wird. Die Operatoren, welche den restlichen Teil der Anfrage repräsentieren, benötigen die entsprechende Variablenbindung während der Ausführung nicht und demzufolge stellt der Operator, welcher g_{inner} repräsentiert, die entsprechende Variablenbindung zwar bereit, aber über den Variablendatenfluss zum g_{Gruppe} repräsentierenden Verbund-Operator wird diese nicht übergeben. Dieses Wissen könnte beispielsweise bei einer verteilten Anfrageausführung genutzt werden.

Jeder Datenfluss ist entweder ein Graphdatenfluss oder ein Variablendatenfluss

$$GF \cup VF = DF$$

und die Mengen der Graph- und der Variablendatenflüsse sind disjunkt

$$GF \cap VF = \emptyset$$

Für die Eigenschaft *output* der Operatoren gelten in Bezug auf die Unterteilung in graph- und variablen-bereitstellende Operatoren die Einschränkungen

$$\begin{array}{ll} \text{i)} & \forall\, o \in GPOP : o.\text{output} = \big(gf_i \mid 1 \leq i < m;\ gf_i \in GF\big) \\ \text{ii)} & \forall\, o \in VPOP : o.\text{output} = \big(vf_i \mid 1 \leq i < n;\ vf_i \in VF\big)\,, \end{array}$$

wobei m und n natürliche Zahlen sind ($m \in \mathbb{N}$; $n \in \mathbb{N}$). Das bedeutet, dass die Ausgangsdatenflüsse von graph-bereitstellenden Operatoren nur Graphdatenflüsse sein können und die Ausgangsdatenflüsse von variablen-bereitstellenden Operatoren nur Variablendatenflüsse. Dementsprechend ist der Wertebereich der Abbildung ProvOp (siehe Definition 3.5 auf Seite 21) in Abhängigkeit vom Definitionsbereich wie folgt eingeschränkt.

$$\begin{array}{ll} \text{i)} & \forall\, d \in GF : \mathsf{ProvOp}(d) \in GPOP \\ \text{ii)} & \forall\, d \in VF : \mathsf{ProvOp}(d) \in VPOP \end{array}$$

Eine weitere Gruppe von Operatoren mit einer herausgehobenen Stellung sind die Operatoren zur Erzeugung des Ergebnisses einer Anfrage. Diese werden in Abschnitt 3.1.9 ab Seite 26 spezifiziert.

3.1.3 Repräsentation von RDF-Graphen

Die einfachsten Repräsentanten der Klasse der graph-bereitstellenden Operatoren sind Operatoren, welche den Zugriff auf einen durch eine URI identifizierten RDF-Graph repräsentieren.

Definition 3.10 Ein *Graphoperator* ist ein graph-bereitstellender Operator. Er repräsentiert den Zugriff auf einen durch eine URI identifizierten RDF-Graph und stellt diesen als Ausgangsdaten bereit. Die Menge GO enthält alle Graphoperatoren eines SQGM, wobei $GO \subseteq GPOP$ gilt. Die URI des bereitgestellten RDF-Graphen ist durch die Eigenschaft

- *uri*

definiert. Für die Eigenschaft *input* gilt die folgende Einschränkung.

$$\forall\, o \in GO : o.\text{input} = \text{undef.}$$

□

Der durch einen Graphoperator bereitgestellte RDF-Graph ist Teil der RDF-Datenmenge, auf der eine SPARQL-Anfrage ausgeführt wird. Die konkrete Art des Zugriffs auf den RDF-Graph ist von der Speicherform des RDF-Graphen abhängig. Beispielsweise könnte er persistent in einer RDF-Datenbank oder flüchtig im Hauptspeicher vorliegen.

Die Einschränkung der Eigenschaft *input* deutet an, dass Graphoperatoren keine eingehenden Datenflüsse haben. Es sei darauf hingewiesen, dass dadurch der Wertebereich der Abbildung $\mathsf{UsingOp}$ (siehe Definition 3.4) wie folgt eingeschränkt ist.

$$\forall\, d \in DF : \mathsf{UsingOp}(d) \notin GO$$

Wenn der Default-Graph einer RDF-Datenmenge aus mehreren einzelnen RDF-Graphen zusammengesetzt ist, was beispielsweise durch eine Reihe von FROM-Klauseln in der SPARQL-Anfrage spezifiziert wird, dann kann der Default-Graph durch den im Folgenden definierten Operator repräsentiert werden.

Definition 3.11 Ein *Graph-Verschmelzungsoperator* (engl.: graph merge operator) ist ein graphbereitstellender Operator und stellt einen durch Verschmelzung mehrerer RDF-Graphen entstandenen RDF-Graph als Ausgangsdaten bereit. Die zu verschmelzenden RDF-Graphen sind die Eingangsdaten des Graph-Verschmelzungsoperators und die exakte Semantik der Verschmelzungsoperation ist in [Hay04, 0.3 Graph Definitions] definiert. Die Menge GMO enthält den Graph-Verschmelzungsoperator eines SQGM, wobei $GMO \subseteq GPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in GMO : o.\text{input} = \left(gf_i \mid 1 \leq i < n;\, gf_i \in GF\right),$$

wobei n eine natürliche Zahl ist ($n \in \mathbb{N}$). □

3.1.4 Repräsentation von einfachen Graphmustern und Wertebeschränkungen

Einfache Graphmuster (siehe Abschnitt 2.3.3 auf Seite 10) und Wertebeschränkungen (siehe Abschnitt 2.3.6 auf Seite 12) werden in SQGM zusammen, in einer Art von Operatoren erfasst.

Definition 3.12 Ein *Graphmuster-Operator* (engl.: graph pattern operator) ist ein variablenbereitstellender Operator dessen Eingabedaten ein RDF-Graph sind. In Bezug auf diesen RDF-Graph ermittelt ein Graphmuster-Operator für eine Menge von Tripelmuster und Wertebeschränkungen alle möglichen Variablenbindungen und stellt diese als Ausgabedaten zur Verfügung. Die Menge GPO enthält alle Graphmuster-Operatoren eines SQGM, wobei $GPO \subseteq VPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in GPO : o.\text{input} = gf\,,$$

wobei gf ein beliebiger Graphdatenfluss ist ($gf \in GF$). Weiterhin haben Graphmuster-Operatoren die folgenden Eigenschaften.

- *triplePatterns* – eine Liste von Tripelmustern, welche während der Ausführung ausgewertet werden
- *constraints* – eine Liste von Wertebeschränkungen, welche während der Ausführung ausgewertet werden
- *contr* – ein Wahrheitswert der angibt, ob die enthaltenen Wertebeschränkungen nachweisbar widersprüchlich sind □

Für die Ermittlung der Variablenbindungen muss ein Graphmuster-Operator zunächst Musterübereinstimmung für die enthaltenen Tripelmuster und Wertebeschränkungen im entsprechenden RDF-Graph finden. Dabei bestimmt er eine Menge von Graphmusterlösungen für diese Graphmuster wie in [PS06, 2.5 Basic Graph Patterns] und [PS06, 3.2 Value Constraints] beschrieben. Die ermittelte Menge von Graphmusterlösungen wird entsprechend der Definition 2.1 (Seite 10) unterteilt. Die dabei entstehenden Teilmengen werden durch jeweils eine Variablenbindung repräsentiert. Alle entsprechend ermittelten Variablenbindung bilden dann die Ausgangsdaten des Graphmuster-Operators.

Die Auswertungsreihenfolge zum Finden der Musterübereinstimmungen für alle betrachteten Tripelmuster und Wertebeschränkungen ist in der SPARQL-Spezifikation [PS06] explizit nicht vorgegeben und darf das Ergebnis der Anfrage nicht beeinflussen. Unter Umständen beeinflusst sie aber die Zeit der Anfrageausführung. Demzufolge ist die Reihenfolge der in *triplePatterns* und *constraints* auftretenden Elemente bei der Ausführung eines Graphmuster-Operators sehr wohl zu beachten. Diese Listen sollten deshalb bei einer Anfrageoptimierung möglichst günstig geordnet werden.

Die Eigenschaft *contr* zeigt einen Widerspruch innerhalb der Wertebeschränkungen des entsprechenden Graphmuster-Operators an, wobei der Wahrheitswert F in diesem Fall nicht bedeutet, dass kein Widerspruch vorhanden ist, sondern dass keiner nachgewiesen werden konnte.

Mit einem Graphmuster-Operator kann man eine Menge von Tripelmustern und Wertebeschränkungen aus demselben Gültigkeitsbereich einer SPARQL-Anfrage modellieren (siehe dazu Abschnitt 4.1.2 ab Seite 41).

3.1.5 Repräsentation von Alternativen

Mittels Graphmuster-Vereinigungen (siehe Abschnitt 2.3.5 auf Seite 11) können in SPARQL Lösungsalternativen beschrieben werden. Die alternativen Graphmuster werden in einem SQGM über Operatoren des im Folgenden definierten Typs miteinander vereinigt.

Definition 3.13 Ein *Vereinigungs-Operator* (engl.: union operator) ist ein variablen-bereitstellender Operator. Er nimmt zwei Mengen von Variablenbindungen als Eingangsdaten entgegen, vereinigt diese Mengen miteinander und stellt das Resultat dieser Vereinigung als Ausgangsdaten bereit. Die Menge UO enthält alle Vereinigungs-Operatoren eines SQGM, wobei $UO \subset VPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in UO : o.\text{input} = (vf_1, vf_2)\,,$$

wobei vf_1 und vf_2 beliebige Variablendatenflüsse sind ($vf_1 \in VF$; $vf_2 \in VF$). □

Mit einem Vereinigungs-Operator kann die Vereinigung von zwei alternativen Graphmustern einer Graphmuster-Vereinigung modelliert werden. Die beiden alternativen Graphmuster werden durch die Operatoren modelliert, welche die beiden Eingangsdatenflüsse des Vereinigungs-Operators bereitstellen. Graphmuster-Vereinigungen mit mehr als zwei vereinigten Graphmustern müssen durch mehrere, vereinigte Vereinigungs-Operatoren modelliert werden.

3.1.6 Repräsentation gruppierter Graphmuster

Jede Graphmuster-Gruppe (siehe Abschnitt 2.3.3 auf Seite 10) einer SPARQL-Anfrage enthält mehrere Graphmuster, welche durch spezielle SQGM-Operatoren repräsentiert werden. Jede solche Gruppierung von Graphmustern wird in einem SQGM durch einen *Verbund* (engl.: join) der zugehörigen Operatoren repräsentiert.

Definition 3.14 Ein *Verbund-Operator* (engl.: join operator) ist ein variablen-bereitstellender Operator. Er nimmt zwei Mengen von Variablenbindungen als Eingangsdaten entgegen, verbindet diese Mengen miteinander und stellt das Resultat als Ausgangsdaten bereit. Bei der Verbund-Operation können je eine Variablenbindung aus jeder Eingangsmenge miteinander verbunden und in die Ergebnismenge übernommen werden, wenn die Werte, welche an die in beiden Eingangsvariablenbindungen erwähnten Variablen, jeweils dieselben sind. Die Menge JO enthält alle Verbund-Operatoren eines SQGM, wobei $JO \subset VPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in JO : o.\text{input} = (vf_{\text{left}}, vf_{\text{right}})\,,$$

wobei vf_{left} und vf_{right} beliebige Variablendatenflüsse sind ($vf_{\text{left}} \in VF$; $vf_{\text{right}} \in VF$). □

Die Einschränkung der Eigenschaft *input* deutet an, dass die Eingabe eines Verbund-Operators ein geordnetes Paar von Variablendatenflüssen ist. Der bereitstellende Operator des ersten kann als linker Operand und der des zweiten als rechter Operand des Verbundes interpretiert werden. Die Verbund-Operation, welche von den Verbund-Operatoren ausgeführt wird, entspricht einem *natürlichen Verbund* (engl.: natural join) der relationalen Algebra (siehe hierzu beispielsweise [GMWU02, 5.2.6 Natural Joins]).

Mit einem Verbund-Operator kann die Gruppierung von zwei Graphmustern modelliert werden. Die beiden gruppierten Graphmuster werden durch die Operatoren modelliert, welche die beiden Eingangsdatenflüsse des Verbund-Operators bereitstellen. Da jeder Verbund-Operator einen binären Verbund repräsentiert, muss eine Graphmuster-Gruppe mit mehr als zwei gruppierten Graphmustern durch mehrere, verbundene Verbund-Operatoren modelliert werden.

3.1.7 Repräsentation optionaler Graphmuster

Optionale Graphmuster (siehe Abschnitt 2.3.4 auf Seite 10) dienen in SPARQL zur Abfrage von unvollständigem Wissen. In einem SQGM werden sie durch optionale Eingabe-Datenflüsse repräsentiert.

Definition 3.15 Jeder Variablendatenfluss (siehe Definition 3.9 auf Seite 22) kann durch die folgende Eigenschaft als optional deklariert werden.

- *optional* – ein Wahrheitswert der angibt, ob die übermittelten Variablenbindungen optional sind

□

3.1.8 Repräsentation von Lösungsanpassungen

Über die in Abschnitt 2.3.8 (Seite 14) vorgestellten Klauseln kann in SPARQL die Menge der Lösungen einer Musterübereinstimmung angepasst werden. Diese Anpassungen werden über Operatoren des im Folgenden definierten Typs modelliert.

Definition 3.16 Ein *Lösungsanpassungs-Operator* (engl.: solution modifier operator) ist ein Variablen-bereitstellender Operator. Er enthält Angaben zur Anpassungen der Lösungsmenge, welche durch die Eigenschaften

- *distinct* – ein Wahrheitswert der angibt, ob die Menge der Resultate disjunkt sein muss, ob also die repräsentierte SELECT-Klausel das Schlüsselwort `DISTINCT` enthält
- *orderBy* – eine geordnete, einer ORDERBY-Klausel entsprechende Liste von Sortierbedingung
- *limit* – die maximale Kardinalität der Lösungsmenge – entspricht dem Wert einer LIMIT-Klausel
- *offset* – der Index der ersten relevanten Lösung – entspricht dem Wert einer OFFSET-Klausel

spezifiziert werden. Der Operator wendet auf eine Menge von Variablenbindungen, welche die Eingangsdaten sind, die entsprechend der enthaltenen Angaben notwendigen Anpassungen an und erzeugt dabei eine Menge von Variablenbindungen als Ausgangsdaten. Auf die konkrete Definition der Semantik der einzelnen Anpassungen sei auf Abschnitt 2.3.8 (Seite 14) verwiesen. Die Menge SMO enthält alle Lösungsanpassungs-Operatoren eines SQGM, wobei $SMO \subset VPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in SMO : o.\text{input} = vf\,,$$

wobei vf ein beliebiger Variablendatenfluss ist ($vf \in VF$). □

Mit einem Lösungsanpassungs-Operator können die Klauseln zur Anpassungen der Lösungsmenge in einer SPARQL-Anfrage modelliert werden. Wenn die repräsentierte Anfrage nur einen Teil der Klauseln zur Anpassung der Lösungsmenge enthält, so ist jeweils nur der Wert der den vorhandenen Klauseln entsprechenden Eigenschaften definiert.

3.1.9 Operatoren zur Ergebniserzeugung

Entsprechend den vier möglichen Ergebnisformen von SPARQL-Anfragen existieren vier Arten von Ergebnis-erzeugenden Operatoren für ein SQGM. Diese sind im Folgenden definiert.

Definition 3.17 Ein *ASK-Ergebnisoperator* (engl.: ask result operator) ist ein Operator der aus einer Menge von Variablenbindungen entsprechend der Semantik einer ASK-Klausel [PS06, 10.5 Asking "yes or no"questions] einen Wahrheitswert erzeugt. Die Menge ARO enthält alle

ASK-Ergebnisoperatoren eines SQGM, wobei $ARO \subset OP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in ARO : o.\text{input} = vf\,,$$

wobei vf ein beliebiger Variablendatenfluss ist ($vf \in VF$). Für die Eigenschaft *output* gilt die folgende Einschränkung.

$$\forall\, o \in GO : o.\text{output} \in \{\mathsf{T}, \mathsf{F}\}$$ □

Definition 3.18 Ein *DESCRIBE-Ergebnisoperator* (engl.: describe result operator) ist ein graph-bereitstellender Operator. Er erzeugt aus einer Menge von Variablenbindungen einen RDF-Graph, der als Ausgangsdaten bereitgestellt wird. Die Erzeugung des RDF-Graph entspricht der Semantik der DESCRIBE-Klausel [PS06, 10.4 Descriptions of Resources]. Die Menge DRO enthält alle DESCRIBE-Ergebnisoperatoren eines SQGM, wobei $DRO \subset GPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in DRO : o.\text{input} = vf\,,$$

wobei vf ein beliebiger Variablendatenfluss ist ($vf \in VF$). Zusätzlich zu den Eigenschaften graph-bereitstellender Operatoren haben DESCRIBE-Ergebnisoperatoren die folgenden Eigenschaften.

- *describedResources* – eine Menge aller zu beschreibender Ressourcen (Variablen und URIs)

□

Definition 3.19 Ein *CONSTRUCT-Ergebnisoperator* (engl.: construct result operator) ist ein graph-bereitstellender Operator. Er erzeugt aus einer Menge von Variablenbindungen einen RDF-Graph, der als Ausgangsdaten bereitgestellt wird. Die Erzeugung des RDF-Graph entspricht der Semantik der CONSTRUCT-Klausel [PS06, 10.3 Constructing an Output Graph]. Die Menge CRO enthält alle CONSTRUCT-Ergebnisoperatoren eines SQGM, wobei $CRO \subset GPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in CRO : o.\text{input} = vf\,,$$

wobei vf ein beliebiger Variablendatenfluss ist ($vf \in VF$). Zusätzlich zu den Eigenschaften graph-bereitstellender Operatoren haben CONSTRUCT-Ergebnisoperatoren die folgende Eigenschaft.

- *template* – ein Graphmuster, welches als Schablone bei der Konstruktion dient

□

Definition 3.20 Ein *SELECT-Ergebnisoperator* (engl.: select result operator) ist ein variablen-bereitstellender Operator. Er erzeugt aus einer Menge von Variablenbindungen eine neue Menge von Variablenbindungen, die als Ausgangsdaten bereitgestellt wird. Die Erzeugung entspricht der Semantik der SELECT-Klausel [PS06, 10.2 Selecting Variables], wobei für die Auswahl die in der Eigenschaft *provVars* enthaltenen Variablen verwendet werden. Die Menge SRO enthält alle SELECT-Ergebnisoperatoren eines SQGM, wobei $SRO \subset VPOP$ gilt. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in SRO : o.\text{input} = vf\,,$$

wobei vf ein beliebiger Variablendatenfluss ist ($vf \in VF$). □

Es sei darauf hingewiesen, dass nach der SPARQL-Spezifikation in ASK-Anfragen keine Angaben zur Anpassung der Lösungsmenge gestattet sind. Übertragen auf das SQGM bedeutet das, der bereitstellende Operator des Eingangsdatenflusses eines ASK-Ergebnisoperators darf kein Lösungsanpassungs-Operator sein. Deshalb muss

$$\forall\, o \in ARO : \mathsf{ProvOp}(o.\text{input}) \notin SMO$$

gelten.

3.1.10 Repräsentation von RDF-Graph-Klauseln und benannten Graphen

Die in Abschnitt 2.3.7 (Seite 12) vorgestellten RDF-Graph-Klauseln können in SPARQL-Anfragen in Form von GRAPH-Klauseln eingesetzt werden, um explizit bestimmte benannte Graphen für die Musterübereinstimmung zu verwenden. Zur ihrer Modellierung ist die im Folgenden definierte Operatorart vorgesehen.

Definition 3.21 Ein *Graphauswahl-Operator* (engl.: graph selection operator) ist ein graph-bereitstellender Operator und stellt einen unter mehreren RDF-Graphen ausgewählten RDF-Graph als Ausgangsdaten bereit. Die auswählbaren RDF-Graphen sind die Eingangsdaten des Graphauswahl-Operators. Die Spezifikation des auszuwählenden RDF-Graphen erfolgt durch Angabe der URI, welche den RDF-Graph identifiziert. Die Menge GSO enthält alle Graphauswahl-Operatoren eines SQGM, wobei $GSO \subseteq GPOP$ gilt. Im Falle der Graphspezifikation mit einer Variable, ist diese Variable durch die Eigenschaft

- *var* – bei Graphspezifikation verwendete Variable

definiert. Für die Eigenschaft *input* gilt die Einschränkung

$$\forall\, o \in GSO : o.\text{input} = \big(gf_i \mid 1 \leq i < n;\, gf_i \in GF\big)\,,$$

wobei n eine natürliche Zahl ist ($n \in \mathbb{N}$). □

Ein Graphauswahl-Operator repräsentiert die Auswahl eines benannten RDF-Graphen, wie von einer RDF-Graph-Klausel mit Graphspezifikation in Form eines leeren Knotens oder einer Variable vorgesehen und mit Hilfe von GRAPH-Klauseln der Form `GRAPH ?var {...}` oder `GRAPH [] {...}` notiert.

3.1.11 Weitere Details

Graph-bereitstellende Operatoren können als Datenquelle angesehen werden, auf welche die variablen-bereitstellenden Operatoren angewandt werden. Für eine einfachere Ausführbarkeit der Operatoren und zur Erleichterung der Analyse eines SQGM, insbesondere bei einem top-down-Vorgehen, dient die im Folgenden induktiv definierte Abbildung.

Definition 3.22 Die Abbildung $\mathsf{DS} : OP \rightarrow \mathfrak{P}(GPOP)$ weist jedem Operator $o \in OP$ eine Menge von graph-bereitstellenden Operatoren als Datenquelle zu. Die tatsächlichen Datenquellen hängen von dem Typ von o ab.

- Fall 1 – Graphoperatoren

  In diesem Fall hat der Operator sich selbst als Datenquelle.

- Fall 2 – Graphmuster-Operatoren

  Die Datenquelle eines Graphmuster-Operators ist der Operator, welcher den Eingangsdatenfluss bereitstellt.

- Fall 3 – Lösungsanpassungs-Operatoren und Operatoren zur Ergebniserzeugung

  In diesem Fall entspricht die Datenquelle des Operators der Datenquelle des Operators, welcher den Eingangsdatenfluss bereitstellt.

- Fall 4 – Graph-Verschmelzungsoperatoren, Graphauswahl-Operatoren, Verbund-Operatoren und Vereinigungsoperatoren

  In diesem Fall wird die Menge der Datenquellen durch die Vereinigung aller Datenquellen der Operatoren, welche die jeweiligen Eingangsdatenflüsse bereitstellend, gebildet.

Es gilt also

$$\mathsf{DS}(o) := \begin{cases} \{o\} & o \in GO \\ \{\mathsf{ProvOp}(o.\text{input})\} & o \in GPO \\ \mathsf{DS}(\mathsf{ProvOp}(o.\text{input})) & o \in SMO \cup SRO \cup DRO \cup CRO \cup ARO \\ \bigcup_{i \in o.\text{input}} \mathsf{DS}(\mathsf{ProvOp}(i)) & o \in JO \cup UO \cup GMO \cup GSO \end{cases}$$

□

3.1.12 Zusammenfassung

Mit Hilfe der, in den vorherigen Abschnitten spezifizierten Bestandteilen eines SQGM kann an dieser Stelle das Modell selbst definiert werden.

Definition 3.23 Das *SPARQL Query Graph Model* (SQGM) ist ein Modell für die Repräsentation von SPARQL-Anfragen. Das SQGM einer Anfrage wird durch ein Tupel

$$(OP, DF, r, dflt, NG)$$

mit den im Folgenden erläuterten Elementen beschrieben.

- OP ist die Menge der Operatoren des SQGM
- DF ist die Menge aller Datenflüsse zwischen den Operatoren in OP
- $r \in SRO \cup DRO \cup CRO \cup ARO \subset OP$ is der Operator, welcher während der Anfrageausführung das Ergebnis der repräsentierten SPARQL-Anfrage erzeugt
- $dflt \in GO \cup GMO \subset OP$ ist der Operator, welcher den Default-Graph der RDF-Datenmenge bereitstellt
- $NG \subseteq GO \subset OP$ ist die Menge von Graphoperatoren, welche die benannten RDF-Graphen der RDF-Datenmenge bereitstellen □

Zur Erzeugung eines SQGM für eine spezielle SPARQL-Anfrage sei auf den Abschnitt 4.1 (ab Seite 38) verwiesen.

Eine sehr wichtige Eigenschaft von zwei SQGMs ist ihre semantische Äquivalenz. Deren Beibehaltung wird von den, im Rahmen dieser Arbeit diskutierten Umformungsregeln verlangt.

Definition 3.24 Zwei SQGMs q und q' sind *semantisch äquivalent*, wenn für jede RDF-Datenmenge D gilt

$$Result_D(q) = Result_D(q') \quad ,$$

wobei $Result_D(m)$ die Menge der Ergebnisse einer, durch das SQGM m repräsentierten SPARQL-Anfrage bezüglich der RDF-Datenmenge D ist. □

Um die formale Definition des SPARQL Query Graph Model für SPARQL-Anfragen abzurunden, gibt die folgende Tabelle noch einmal einen referenzartigen Überblick über alle Typen von Operatoren.

Operatortyp	Def.	Seite	Menge	Klasse	Anfrageteil	Eigenschaften
Graphoperator	3.10	23	*GO*	graph-bereitstellend	erwähnter RDF-Graph, Teil der RDF-Datenmenge	*output*; *uri*
Graph-Verschmelzungsoperator	3.11	24	*GMO*	graph-bereitstellend	zusammengesetzter Default-Graph	*input*; *output*
Graphauswahl-Operator	3.21	28	*GSO*	graph-bereitstellend	RDF-Graph-Klausel mit leerem Knoten oder Variable als Graphspezifikation	*input*; *output*; *var*
Graphmuster-Operator	3.12	24	*GPO*	variablen-bereitstellend	Tripelmuster und Wertebeschränkungen im selben Gültigkeitsbereich	*input*; *output*; *provVars*; *triplePatterns*; *constraints*; *contr*
Vereinigungs-Operator	3.13	25	*UO*	variablen-bereitstellend	Graphmuster-Vereinigung (UNION-Klausel)	*input*; *output*; *provVars*
Verbund-Operator	3.14	25	*JO*	variablen-bereitstellend	(Teil einer) Graphmuster-Gruppe	*input*; *output*; *provVars*
Lösungsanpassungs-Operator	3.16	26	*SMO*	variablen-bereitstellend	Anpassungen der Lösungsmenge (ORDERBY-, LIMIT- und OFFSET-Klausel)	*input*; *output*; *provVars*; *distinct*; *orderBy*; *limit*; *offset*
SELECT-Ergebnisoperator	3.20	27	*SRO*	ergebnis-erzeugend; variablen-bereitstellend	SELECT-Klausel	*input*; *output*; *provVars*
DESCRIBE-Ergebnisoperator	3.18	27	*DRO*	ergebnis-erzeugend und graph-bereitstellend	DESCRIBE-Klausel	*input*; *describedResources*; *output*
CONSTRUCT-Ergebnisoperator	3.19	27	*CRO*	ergebnis-erzeu- gend und graph-bereitstellend	CONSTRUCT-Klausel	*input*; *output*; *template*
ASK-Ergebnisoperator	3.17	26	*ARO*	ergebnis-erzeugend	ASK-Klausel	*input*

Tabelle 3.1: Überblick über alle SQGM-Operatoren

3.2 Grafische Darstellung eines SQGM

SQGMs sind gerichtete Graphen mit Operatoren als Knoten und Datenflüssen als gerichtete Kanten. Hierbei verbindet eine Kante zum Datenfluss $d \in DF$ die Knoten, welche $\mathsf{ProvOp}(d)$ und $\mathsf{UsingOp}(d)$ repräsentieren. In einer grafischen Darstellung von SQGM-Graphen werden die Knoten in Form von Boxen visualisiert. Diese Boxen können zusätzliche Elemente beinhalten, welche von der Art des repräsentierten Operators und dessen Eigenschaften abhängen. Diese Elemente werden im Folgenden genauer spezifiziert und danach in einer abschließenden Tabelle zusammengefasst.

Boxen die variablen-bereitstellende Operatoren (siehe Definition 3.7 auf Seite 22) repräsentieren, werden in einen Kopf und Rumpf unterteilt. Der Kopfteil enthält eine Liste der Variablen, für die vom entsprechenden Operator Bindungen bereitgestellt werden, also den Inhalt der Eigenschaft *provVars*. Entsprechend werden die Kanten, welche Variablendatenflüsse (siehe Definition 3.9 auf Seite 22) repräsentieren, mit einer Liste von übergebenen Variablen, also dem Inhalt der Eigenschaft *vars* beschriftet. Die vermeintlich jeweils doppelte Aufzählung der Variablen ermöglicht die Modellierung des Falles, dass nicht alle angebotenen Variablenbindungen von einem benutzenden Operator während der Ausführung benötigt werden. Die Variablendatenfluss-Kanten erhalten zusätzlich das Schlüsselwort `optional`, wenn die Eigenschaft *optional* (siehe Definition 3.15 auf Seite 26) des zugehörigen Variablendatenflusses T ist.

Bei den Boxen für Graphmuster-Operatoren (siehe Definition 3.12 auf Seite 24) werden im Rumpf die einzelnen Elemente der Eigenschaften *triplePatterns* und *constraints*, also die einzelnen Tripelmuster und Wertebeschränkungen aufgelistet. Der Rumpf von Boxen für Vereinigungs-Operatoren (siehe Definition 3.13 auf Seite 25) enthält lediglich das Schlüsselwort `UNION`. Die beiden Kanten zu den Eingangsdatenflüssen werden zusätzlich, entsprechend ihrem Index in der Eigenschaft *input* mit einer Eins oder Zwei beschriftet. Analog enthalten die Rümpfe von Boxen für Verbund-Operatoren (siehe Definition 3.14 auf Seite 25) das Schlüsselwort `JOIN` und die eingehenden Kanten werden zusätzlich mit `left` und `right` beschriftet. Boxen für Lösungsanpassungs-Operatoren (siehe Definition 3.16 auf Seite 26) enthalten im Rumpf die Eigenschaften *orderBy*, *limit* und *offset* mit den zugehörigen Werten und zusätzlich das Schlüsselwort `DISTINCT` wenn die entsprechende Eigenschaft *distinct* T ist.

Der Rumpf von Boxen für SELECT-Ergebnisoperatoren (siehe Definition 3.20 auf Seite 27) enthält das Schlüsselwort `SELECT`. Die anderen Operatoren zur Ergebniserzeugung sind keine variablen-bereitstellenden Operatoren. Die sie darstellenden Boxen sind einfach. Die Boxen für DESCRIBE-Ergebnisoperatoren (siehe Definition 3.18 auf Seite 27) enthalten das Schlüsselwort `DESCRIBE` gefolgt von einer Liste der zu beschreibenden Ressourcen, entsprechend der Eigenschaft *describedResources*. CONSTRUCT-Ergebnisoperatoren (siehe Definition 3.19 auf Seite 27) werden durch Boxen mit dem Schlüsselwort `CONSTRUCT` und der Graphmuster-Schablone aus der Eigenschaft *template* dargestellt. Die Boxen für ASK-Ergebnisoperatoren (siehe Definition 3.17 auf Seite 26) enthalten lediglich das Schlüsselwort `ASK`.

Graphoperatoren (siehe Definition 3.10 auf Seite 23) werden durch einfache Boxen dargestellt, welche die URI des repräsentierten Graphen enthalten. Entsprechend der Einschränkung der Eigenschaft *input* von Graphoperatoren haben die zugehörigen Boxen keine eingehenden Kanten. Entspricht der, durch einen Graphoperator repräsentierte RDF-Graph dem Default-Graph, so wird die Box des entsprechenden Graphoperators mit dem Schlüsselwort `DEFAULT` beschriftet.

Zur grafischen Darstellung eines Graph-Verschmelzungsoperators (siehe Definition 3.11 auf Seite 24) wird eine einfache Box mit dem Schlüsselwort `MERGE` verwendet. Die Box eines Graph-Verschmelzungsoperators, welcher den Default-Graph repräsentiert, wird mit dem Schlüsselwort `DEFAULT` beschriftet.

Bei der grafischen Darstellung von Graphauswahl-Operatoren muss zwischen den beiden repräsentierten GRAPH-Klausel-Formen, also den Graphspezifikationsarten leerer Knoten oder Variable, unterschieden werden. Im ersten Fall kann der Graphauswahl-Operator durch eine einfache Box mit dem Schlüsselwort `SELECTION` dargestellt werden. Im zweiten Fall besteht die Box aus einem Kopf und einem Rumpf. Der Kopf enthält die zur Bindung an die Graph-URI vorgesehene Variable, welche durch die Eigenschaft *var* spezifiziert ist, und der Rumpf enthält das Schlüsselwort `SELECTION`.

Beispiel 3.1 *Die folgende SPARQL-Anfrage enthält unter anderem ein optionales Graphmuster, eine Angabe zur Anpassung der Lösungsmenge und die Angabe von zwei RDF-Graphen, welche für den Default-Graphen verschmolzen werden müssen. Sie kann durch das in Abbildung 3.1 dargestellte SQGM repräsentiert werden.*

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
FROM    <http://example.org/foaf/aliceFoaf>
FROM    <http://example.org/foaf/bobFoaf>
WHERE {
        ?x foaf:name ?name .
        OPTIONAL { ?x foaf:mbox ?mbox }
}
ORDER BY ?name
```

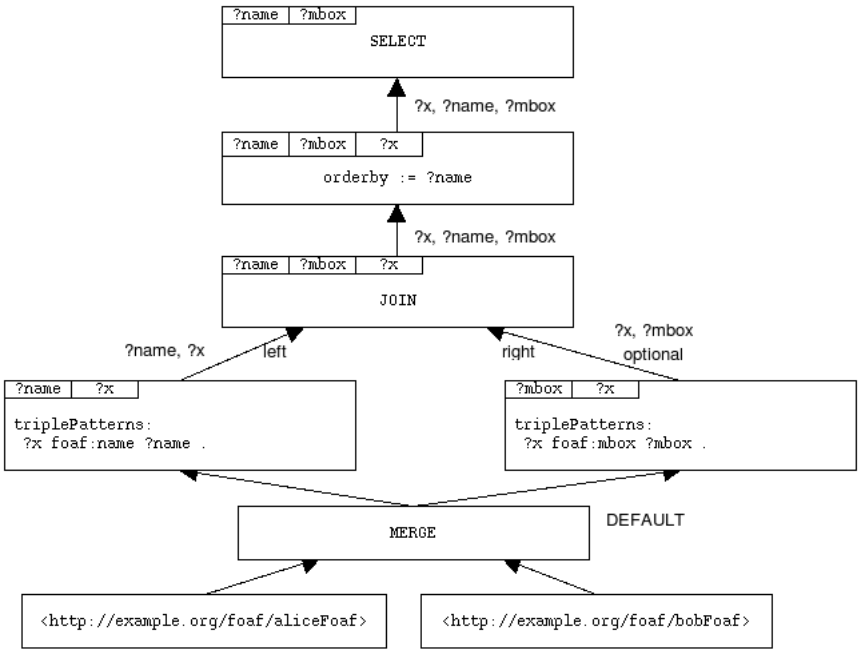


Abbildung 3.1: SQGM für eine Auswahl-Anfrage mit optionalem Graphmuster, Lösungsmengenanpassung und zusammengesetztem Default-Graph □

Beispiel 3.2 *Die Abbildung 3.2 stellt ein SQGM dar, welches die folgende Konstruktionsanfrage repräsentiert. Sie enthält unter anderem eine RDF-Graph-Klausel mit zugehöriger Aufzählung von benannten Graphen und eine Wertebeschränkung.*

```
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
PREFIX app: <http://example.org/ns#>
CONSTRUCT { ?s ?p ?o }
FROM NAMED <http://example.org/data/2004/>
FROM NAMED <http://example.org/data/2005/>
FROM NAMED <http://example.org/data/2006/>
WHERE {
        GRAPH ?g { ?s ?p ?o } .
        ?g dc:publisher <http://www.w3.org/> ;  ?g dc:date ?date .
        FILTER ( app:customDate(?date) > "2005-02-28T00:00:00Z"^^xsd:dateTime )
}
```

□

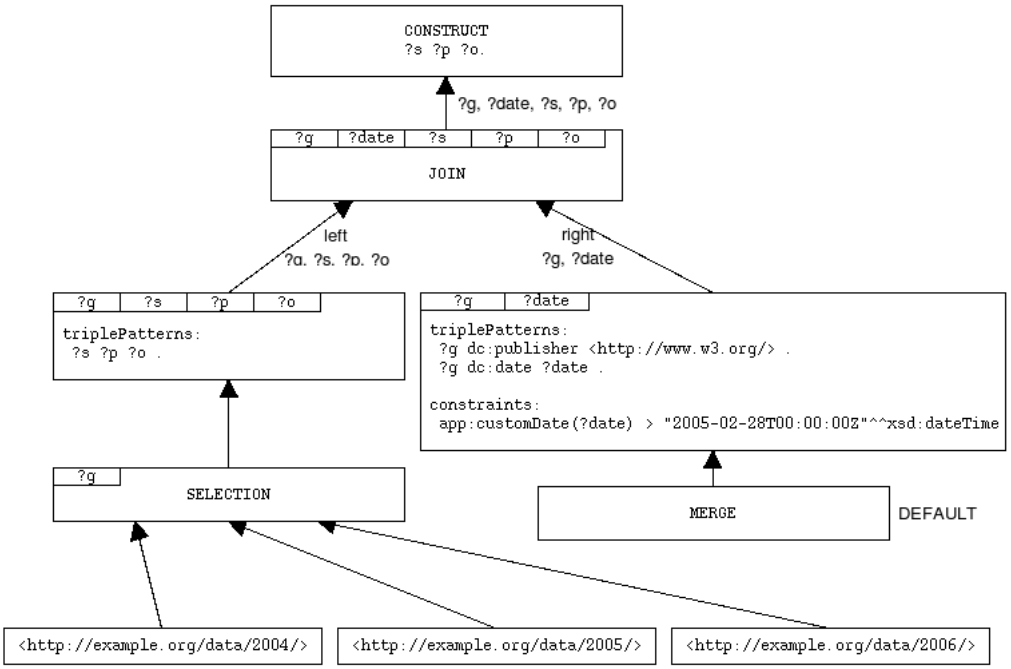


Abbildung 3.2: Ein SQGM mit CONSTRUCT-Box für eine Anfrage mit RDF-Graph-Klausel und Wertebeschränkung

In der Tabelle 3.2 (Seite 34) sind die Eigenschaften der Boxen für alle Typen von Operatoren zusammengefasst.

3.3 Anfragekompilierung auf Basis des SQGM

Das SQGM dient als interne Repräsentation einer SPARQL-Anfrage im Rahmen ihrer Bearbeitung. Das Vorgehen bei der Anfragebearbeitung kann sich an den, in Abschnitt 2.4.1 (ab Seite 15) vorgestellten Phasen *Anfragekompilierung* (engl.: query compilation) und *Anfrageausführung* (engl.: query execution) zur Bearbeitung von Datenbankanfragen orientieren. In diesem Abschnitt werden die verschiedenen Schritte eines möglichen Vorgehens innerhalb der ersten Hauptphase, der Anfragekompilierung, auf der Basis des SQGM skizziert. Es werden jedoch keine konkreten Vorschläge für Strategien zur Optimierung oder Realisierungsmethoden diskutiert oder mögliche Kostenmodelle vorgestellt.

Analog zur Anfragebearbeitung in einem DBMS ist auch bei der Bearbeitung von SPARQL-Anfragen das Ziel der Anfragekompilierung die Entwicklung eines möglichst optimalen Ausführungsplans. Dafür kann die Anfragekompilierung in die Schritte Anfrageübersetzung und -vereinfachung, logische Optimierung und physische Optimierung mit kostenbasierter Auswahl zerlegt werden (siehe Abbildung 3.3). Diese Zerlegung ist ebenfalls an die DBMS-Anfragebearbeitung angelehnt. Im Folgenden wird auf die einzelnen Schritte näher eingegangen, wobei ein besonderes Augenmerk auf die RDF- und SPARQL-spezifischen Details gelegt wird.

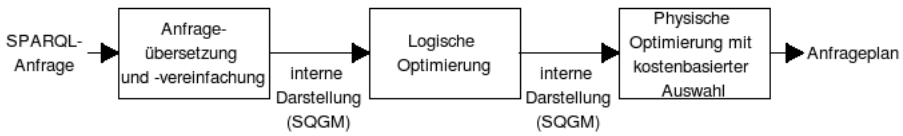


Abbildung 3.3: Schritte bei der Anfragekompilierung in Anlehnung an [GS05]

Operatortyp	Kopf	Rumpf
Graphoperator	URI des repräsentierten Graphen (entsprechend der Eigenschaft *uri*)	leer
Graph-Verschmelzungsoperator	nicht vorhanden	Schlüsselwort `MERGE`
Graphauswahl-Operator	Variable der Graphspezifikation (entsprechend der Eigenschaft *var*) oder nicht vorhanden	Schlüsselwort `SELECTION`
Graphmuster-Operator	Variablenliste (entsprechend der Eigenschaft *provVars*)	eine Liste von Tripelmustern und Wertebeschränkungen (entsprechend den Eigenschaften *triplePatterns* und *constraints*)
Vereinigungs-Operator	Variablenliste (entsprechend der Eigenschaft *provVars*)	Schlüsselwort `UNION`
Verbund-Operator	Variablenliste (entsprechend der Eigenschaft *provVars*)	Schlüsselwort `JOIN`
Lösungsanpassungs-Operator	Variablenliste (entsprechend der Eigenschaft *provVars*)	Eigenschaften *orderBy*, *limit* und *offset* und eventuell Schlüsselwort `DISTINCT` (in Abhängigkeit der Eigenschaft *distinct*)
SELECT-Ergebnisoperator	Variablenliste (entsprechend der Eigenschaft *provVars*)	Schlüsselwort `SELECT`
DESCRIBE-Ergebnisoperator	nicht vorhanden	Schlüsselwort `DESCRIBE` mit Symbol * oder Liste zu beschreibender Ressourcen (entsprechend der Eigenschaft *describedResources*)
CONSTRUCT-Ergebnisoperator	nicht vorhanden	Schlüsselwort `CONSTRUCT` mit Konstruktionsschablone (entsprechend der Eigenschaft *template*)
ASK-Ergebnisoperator	nicht vorhanden	Schlüsselwort `ASK`

Tabelle 3.2: Eigenschaften von Boxen zur Visualisierung von SQGM-Operatoren

3.3.1 Anfrageübersetzung und -vereinfachung

Die Anfragekompilierung beginnt mit der Anfrageübersetzung und -vereinfachung, deren Teilschritte in Abbildung 3.4 dargestellt sind.

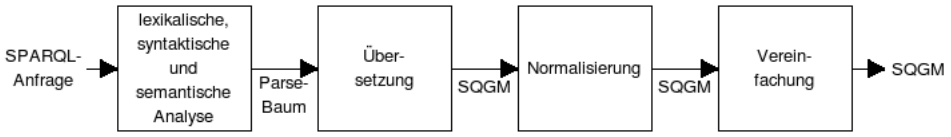


Abbildung 3.4: Teilschritte der Anfrageübersetzung und -vereinfachung

Hierbei wird aus einer gegebenen SPARQL-Anfrage ein SQGM erzeugt, welches als interne Darstellung der Anfrage dem Operatorbaum einer DBMS-Anfragebearbeitung entspricht. Zur Erzeugung muss die als Zeichenkette gegebene Anfrage lexikalisch, syntaktisch und semantisch analysiert und nach fest definierten Regeln in ein initiales SQGM umgewandelt werden. Bis auf die semantische Analyse entspricht dieser Teilschritt der Anfrageübersetzung in einem DBMS, welche beispielsweise in [GMWU02, Seite 788ff.] beschrieben ist. Im Rahmen der semantischen Analyse von SPARQL-Anfragen ist zu prüfen, ob die RDF-Datenmenge, auf welcher die Anfrage ausgeführt wird, für die Anfragebearbeitung vollständig zur Verfügung steht oder zusammengestellt werden kann.

In den weiteren Teilschritten kann das erste Modell durch Anwendung von Transformationsregeln normalisiert und vereinfacht werden. Die Normalisierung führt zu einer, für die weitere Verarbeitung, standardisierten Form des SQGM. Wie die Normalform eines SQGM aussieht, wäre natürlich zu definieren. Ein möglicher Aspekt eines normalisierten SQGM wäre beispielsweise die Freiheit von leeren Graphmuster-Operatoren, welche eventuell wegen, in SPARQL syntaktisch möglicher, leerer Graphmuster modelliert wurden. Möglichkeiten zur Vereinfachung sind die Befreiung eines SQGM von Redundanzen, welche durch gleiche Teilgraphmuster innerhalb des Graphmusters der SPARQL-Anfrage entstehen könnten, und die Kennzeichnung und Behandlung von syntaktisch festgestellbaren Widersprüchen innerhalb des Graphmusters. Die einzelnen Teilschritte der Anfrageübersetzung und -vereinfachung auf Basis des SQGM werden in Kapitel 4 (ab Seite 38) detailiert diskutiert.

3.3.2 Logische Optimierung

Während der Optimierungsphase wird für ein SQGM ein möglichst effizienter Ausführungsplan bestimmt, wobei ein entsprechendes Optimalitätskriterium definiert werden muss. Der tatsächlichen Planerzeugung im Rahmen der physischen Optimierung geht hierbei die logische Optimierung voraus. Sie basiert auf der Anwendung von Transformationsregeln zur Restrukturierung des SQGM, wobei alle Regeln die Semantik der repräsentierten Anfrage beibehalten müssen. Die Umformungen des SQGM dürfen also keinen Einfluss auf die Lösungsmenge der Anfrage haben. Die logische Optimierung kann verschiedene der folgenden Ziele anstreben, welche im Folgenden näher erläutert werden.

- effizientere Anfrageausführung
- Erhöhung der Möglichkeiten zur Planerzeugung

Die Verfolgung der angegebenen Ziele sind nur als mögliche Aufgaben der logischen Optimierung zu verstehen und ein konkretes Anfragebearbeitungssystem muss sich nicht auf beide konzentrieren.

Das eine Ziel ist die Verbesserung des SQGM, welches hier als logischer Anfrageplan verstanden wird, für eine optimalere Anfrageausführung. Die dabei angewandten Strategien zur Umformung des SQGM, also die Bestimmung einer konkreten Kaskade von Transformationsregeln, stützen sich auf Heuristiken. In Kapitel 6 (ab Seite 90) werden einige mögliche Heuristiken formuliert

und untersucht. Es ist davon auszugehen, dass die Gültigkeit der Heuristiken stark von der Art des Modells abhängt, in welchem die abgefragte RDF-Datenmenge vorliegt. So ist es denkbar, dass unterschiedliche oder sogar gegensätzliche Heuristiken angewandt werden müssen, wenn die Daten in einer Datenbank liegen, im Speicher gehalten werden, in einem Netzwerk verteilt sind oder einer logischen Folgerung über eine Basisdatenmenge entstammen. Auch wenn Heuristiken in den meisten Fällen die Anfragen verbessern oder zumindest nicht verschlechtern, sollte erwähnt werden, dass es auch Ausnahmen geben kann.

Das andere mögliche Ziel der Restrukturierung ist die Verbesserung des SQGM in Hinblick auf die Planerzeugung. Dabei wird das SQGM in ein alternatives SQGM transformiert, welches mehr Möglichkeiten zur Erzeugung von Plänen bietet. Die in diesem Fall angewandten Umformungsstrategien beruhen auf Regeln, welche sicherlich, analog zu den Heuristiken, von der Art der Modelle für die Datenmengen abhängen. Beispielsweise birgt die Verschmelzung von Graphmuster-Operatoren eine höhere Anzahl an möglichen Datenbankanfragen, wenn die RDF-Daten von einem DBMS verwaltet werden, und eröffnet zusätzliche Möglichkeiten zur Bestimmung von Musterübereinstimmungen in mehreren Graphmustern gleichzeitig. Bei verteilten Datenmengen wäre es andererseits sinnvoll, die Anfrageausführung zu unterteilen und eine aggressive Verschmelzungsstrategie demzufolge eher kontraproduktiv. Zusätzlich müssen im Rahmen der Verbesserung des SQGM für die physische Optimierung die unterschiedlichen Strategien zur Bestimmung der Lösungen von Graphmustern in Betracht gezogen werden. So ist eine spezielle Form eines SQGM besser für die Tiefensuche geeignet und eine andere für den Kompositionsansatz (siehe Abschnitt 2.4.2 ab Seite 17). Ein Anfrageeausführungssystem, das beide Strategien beherrscht oder diese sogar miteinander kombiniert, könnte bereits während der logischen Optimierung verschiedene logische Anfragepläne, also alternative SQGMs, erzeugen und der physischen Optimierung zur Verfügung stellen.

Zusammenfassend kann gesagt werden, dass für die logische Optimierung Kenntnisse über die Art der abgefragten Datenmenge und unter Umständen auch über die Möglichkeiten der physischen Optimierung und der Anfrageausführung, bezüglich der einsetzbaren Strategien zur Lösungsbestimmung, benötigt werden. Eine Kenntnis des Schemas der Daten könnte ebenfalls nützlich sein, falls Transformationsregeln exisitieren, welche diese ausnutzen. Hingegen sind Informationen zu den konkreten Daten, deren Menge und Verteilung, der tatsächlichen Organisation der verwendeten Modelle und eventuell existierender Indexstrukturen im Rahmen der logischen Optimierung nicht notwendig.

3.3.3 Physische Optimierung mit kostenbasierter Auswahl

Bei der physischen Optimierung mit kostenbasierter Auswahl werden aus dem SQGM verschiedene physische Ausführungspläne erzeugt und der in Bezug auf ein Kostenmodell optimalste ausgewählt. Diese Auswahl kann durch eines der Verfahren implementiert werden, die im Rahmen der Optimierung von Datenbankanfragen bereits detailiert untersucht und beschrieben wurden. Beispiele für mögliche Auswahlverfahren sind die vollständige Suchraumexpansion oder eine iterative Näherung, welche unter anderem in [GMWU02, Seite 842ff.] vorgestellt werden. Die Anpassung dieser Verfahren an die Optimierung von SPARQL-Anfragen dürfte lediglich in der Verwendung spezieller Kostenmodelle liegen, welche für RDF-Datenmengen entwickelt werden und auf die konkreten Speicher- und Zugriffsmodelle abgestimmt sind. Eine Beschreibung oder Entwicklung solcher Kostenmodelle liegt jedoch nicht im Rahmen dieser Arbeit.

Die Erzeugung eines ausführbaren, physischen Plans aus einem SQGM beinhaltet die Festlegung einer oder mehrerer Strategien zur Bestimmung der Lösungen von Graphmustern, eine Umwandlung der logischen Operatoren des SQGM in physische und die Festlegung einer Ausführungsreihenfolge. Diese Schritte sind von der Art der Speicher- und Zugriffsmodelle für die RDF-Datenmengen und eventuell existierenden Index-Strukturen abhängig. Zusätzlich könnten Kenntnisse über das Schema der Daten in die Planerzeugung einfließen.

Die möglichen Strategien zur Lösungsbestimmung basieren auf dem logischem Schließen oder verfolgen einen prozeduralen Ansatz mit Tiefensuche oder den Kompositionsansatz (siehe Abschnitt 2.4.2 ab Seite 17). Die Anwendung der Strategien ist nicht als exklusiv zu verstehen. So

ist es beispielsweise möglich, dass einzelne Teile eines SQGM mittels des Kompositionsansatzes bearbeitet werden und die restliche Ausführung auf der Tiefensuche beruht. Im Rahmen der verteilten Anfrageausführung scheint insbesondere dieser hybride Ansatz sinnvoll. Die Entscheidung für ein oder mehrere Strategien liegt im Ermessen der automatisierten Planerzeugung und ist zunächst von den Möglichkeiten der Ausführungskomponente und dann auch von den erwähnten Kenntnissen über Speicher- und Zugriffsmodelle, Index-Strukturen und Schemata abhängig. Andererseits beeinflusst diese Entscheidung natürlich auch die folgenden Schritte der Erzeugung des Ausführungsplans.

Nach der Auswahl der Lösungsstrategie muss das SQGM in einen physischen Plan umgewandelt werden, wobei dessen physische Operatoren aus den logischen durch Zuordnung von konkreten Realisierungsmethoden entstehen. Für die einzelnen Operatoren kommen verschiedene solcher Methoden in Frage. Diese werden hier nicht diskutiert. Es sei jedoch darauf hingewiesen, dass sie von der festgelegten Lösungsstrategie und den oben erwähnten Informationen abhängig sind. Für den Datenzugriff könnten Index-Strukturen, wie in [HD05] oder [RvGB03] vorgestellt, verwendet werden. Die Übereinstimmungen von Graphmustern könnten nach Algorithmen zur Bestimmung der Äquivalenz von RDF-Graphen [Car02] oder durch logische Schlussverfahren [dBFT05] bestimmt werden. Zusätzlich könnten hier Schema-Kenntnisse ausgenutzt werden. An dieser Stelle sei auch darauf hingewiesen, dass über Datenflüsse verknüpfte Operatoren durch *datenstrombasierte Verarbeitung* (engl.: pipelining) zusammengefasst werden könnten.

Die Ausführungsreihenfolge der Operatoren, insbesondere in Verbund- oder Vereinigungsbäumen, kann einen wesentlichen Einfluss auf die zu optimierenden Aspekte der Anfrageausführung haben. Die Verknüpfung der Operatoren eines SQGM beinhaltet implizit eine Ausführungsreihenfolge. Im Rahmen der Erzeugung des physischen Plans sollte diese jedoch explizit festgelegt werden. Hierfür können die variablen-bereitstellenden Operatoren entweder nach einer bestimmten Strategie nummeriert werden oder es kann die Besuchsreihenfolge während einer Tiefen- oder Breitensuche als Ausführungsreihenfolge festgesetzt werden. Dann müsste das SQGM eventuell umgruppiert werden, um die gewünschte Sequenz zu erhalten. In beiden Fällen könnten für die Festlegung Kenntnisse über die Daten, wie die Anzahl an Individuen einer Klasse oder die Selektivität einzelner Klassen bezüglich einer bestimmten Wertebeschränkung, angewandt werden.

Kapitel 4

Anfrageübersetzung und -vereinfachung auf Basis des SQGM

Der erste Schritt der Anfragekompilierung einer SPARQL-Anfrage ist die Anfrageübersetzung und -vereinfachung (siehe Abbildung 3.3 auf Seite 33). Auf diesen Schritt, im Rahmen einer Anfragebearbeitung auf Basis des SQGM, geht dieses Kapitel ein. Wie in Abschnitt 3.3.1 (ab Seite 35) erwähnt, unterteilt sich die Anfrageübersetzung und -vereinfachung in die Analyse, Übersetzung, Normalisierung und Vereinfachung. Entsprechend dieser Teilschritte ist das Kapitel gegliedert, wobei auf die Analyse nicht näher eingegangen wird, da für die lexikalische und syntaktische Analyse problemlos Konzepte des Compilerbau angewandt werden können und die semantische Analyse von SPARQL-Anfragen ausserhalb des Rahmens dieser Arbeit liegt.

Der erste Abschnitt spezifiziert ein Vorgehen zur Übersetzung einer SPARQL-Anfrage in ein SQGM und die darauf folgenden beiden Abschnitte beschäftigen sich mit der Normalisierung des initialen Modells und dessen Vereinfachung. Eine konkrete SPARQL-Anfrage begleitet die Vorgehensbeschreibungen beispielhaft.

4.1 Erstellung eines SQGM

Dieser Abschnitt beschreibt ein mögliches Vorgehen bei der Erstellung eines SQGM zu einer gegebenen SPARQL-Anfrage. Dabei wird auf die einzelnen Teile einer Anfrage und deren Repräsentationsmöglichkeiten eingegangen. Das beschriebene Vorgehen entwickelt das SQGM von unten nach oben. Hierbei werden zu Beginn die Graphoperatoren und am Ende die Operatoren zur Ergebniserzeugung erstellt.

Eine SPARQL-Anfrage ist laut der SPARQL-Spezifikation [PS06] ein Tupel (GP, DS, SM, R) bestehend aus einem Graphmuster GP, einer RDF-Datenmenge DS, einer Menge von Lösungsanpassungen SM und einer Ergebnisform R. Bei der Erstellung eines SQGM für eine solche Anfrage wird zunächst ein leeres SQGM (siehe Definition 3.23 auf Seite 29) mit

$$\begin{aligned} OP &:= \emptyset \\ DF &:= \emptyset \\ r &:= \text{undef.} \\ dflt &:= \text{undef.} \\ NG &:= \emptyset \end{aligned}$$

erzeugt und dieses dann in den Schritten

1. Modellierung der RDF-Datenmenge DS,
2. Modellierung des Graphmusters GP,

3. Modellierung der Menge von Lösungsanpassungen SM und
4. Modellierung der Ergebnisform R

erweitert. Die folgenden Unterabschnitte beschreiben diese Schritte.

4.1.1 Modellierung der RDF-Datenmenge

Die RDF-Datenmenge DS einer SPARQL-Anfrage ist eine Menge, bestehend aus dem Default-Graph G und keinem, einem oder mehreren benannten Graphen $\big(\langle u_j \rangle, G_j\big)$.

$$DS := \Big\{G, \big(\langle u_1 \rangle, G_1\big), \big(\langle u_2 \rangle, G_2\big), \cdots \big(\langle u_n \rangle, G_n\big)\Big\}$$

Während der Konstruktion des SQGM ist bei der Modellierung der RDF-Datenmenge DS zu unterscheiden, ob DS separat der Anfragebearbeitungskomponente übergeben wird oder nicht. Im ersten Fall werden, wie in der SPARQL-Spezifikation festgelegt, eventuell in der Anfrage vorhandene FROM und FROM NAMED Klauseln ignoriert. Im zweiten Fall ist DS zum Zeitpunkt der Erzeugung des SQGM nicht explizit gegeben und muss durch die Angaben in den FROM und FROM NAMED Klauseln generiert werden.

Wenn DS explizit gegeben ist, dann wird für G ein Graphoperator $o \in GO$ mit

$$o.\text{uri} := \text{undef.}$$
$$o.\text{output} := \text{undef.}$$

erzeugt und dem SQGM als Default-Graph repräsentierender Operator hinzugefügt.

$$OP := OP \cup \{o\}$$
$$dflt := o$$

Für jeden benannten Graph $\big(\langle u_j \rangle, G_j\big)$ wird jeweils ein Graphoperator $o_j \in GO$ mit

$$o_j.\text{uri} := u_j$$
$$o_j.\text{output} := \text{undef.}$$

erzeugt und dem SQGM hinzugefügt.

$$OP := OP \cup \{o_j\}$$
$$NG := NG \cup \{o_j\}$$

In dem zweiten Fall, wenn also die RDF-Datenmenge DS nicht explizit gegeben ist, muss sie durch die Angaben in den FROM und FROM NAMED Klauseln erzeugt werden, was durch die im Folgenden erstellten Operatoren repräsentiert wird. Seien m die Anzahl der FROM und n die Anzahl der FROM NAMED Klauseln und dg_i und ng_j die m bzw. n in den Klauseln angegebenen URIs. Für jeden benannten Graph wird, wie im ersten Fall, jeweils ein Graphoperator $o_j \in GO$ mit

$$o_j.\text{uri} := ng_j$$
$$o_j.\text{output} := \text{undef.}$$

erzeugt und dem SQGM hinzugefügt.

$$OP := OP \cup \{o_j\}$$
$$NG := NG \cup \{o_j\}$$

Bei der Konstruktion des Operators für den Default-Graph werden im ersten Schritt Graphoperatoren für die einzelnen Teilgraphen bestimmt oder erzeugt. Hierzu wird NG für jeden dg_i nach

einem o_i mit $o_i.\text{uri} = dg_i$ durchsucht. Wird ein solcher o_i nicht gefunden, muss er mit den Eigenschaften

$$o_i.\text{uri} := dg_i$$
$$o_i.\text{output} := \text{undef.}$$

erstellt und dem SQGM hinzugefügt werden.

$$OP := OP \cup \{o_i\}$$

Im zweiten Schritt müssen wiederum zwei Fälle unterschieden werden. Im Fall, dass $m = 1$ gilt, wird o_1 als Default-Graph repräsentierender Operator gesetzt.

$$dflt := o_1$$

Im anderen Fall, also wenn $m \neq 1$ ist, wird ein Graph-Verschmelzungsoperator $o \in GMO$ mit den Eigenschaften

$$o.\text{input} := (gf_1, \cdots, gf_m)$$
$$o.\text{output} := ()$$

erzeugt, dem SQGM als Default-Graph repräsentierender Operator hinzugefügt

$$OP := OP \cup \{o\}$$
$$dflt := o$$

und über m neu zu erstellende Graphdatenflüsse $gf_i \in GF$

$$DF := DF \cup \{gf_1, \cdots, gf_m\}$$

mit den einzelnen teilgraph-repräsentierenden Operatoren verbunden

$$\forall\, i \in \{1, \cdots, m\} : o_i.\text{output} := o_i.\text{output} \circ gf_i$$

wobei $L \circ e$ das Anhängen des Elementes e an die Liste L repräsentiert.

Beispiel 4.1 *Mit der folgenden SPARQL-Anfrage kann eine alphabetisch absteigend sortierte Liste der Bergketten bestimmt werden, in denen Berge liegen, welche über 4000 Meter hoch sind und in dem durch die URI* `http://www.example.org/mountains.rdf` *repräsentierten RDF-Graph erwähnt werden.*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT DISTINCT ?r
FROM http://www.example.org/mountains.rdf
WHERE {
   ?m exm:range ?r ;
      exm:height ?h .
   FILTER ( ?h > 4000 ).
}
ORDER BY DESC(?r)
```

Wenn die URI `http://www.example.org/mountains.rdf` *den in Beispiel 2.11 (ab Seite 8) dargestellten RDF-Graph repräsentiert, liefert diese Anfrage das folgende Resultat.*

?r
http://www.example.org/mountains/PennineAlps
http://www.example.org/mountains/BerneseAlps

Der Default-Graph dieser Anfrage wird in einem SQGM durch einen Operator $dflt \in GO$ mit den Eigenschaften

$$dflt.\text{uri} = \texttt{http://www.example.org/mountains.rdf}$$
$$dflt.\text{input} = ()$$

repräsentiert. □

4.1.2 Modellierung des Graphmusters

Für die verschiedenen Arten von Graphmustern (siehe Abschnitt 2.3.2 ab Seite 9) müssen, wie im Folgenden angegeben, unterschiedliche SQGM-Operatoren erzeugt und verknüpft werden. Einige Graphmuster enthalten wiederum weitere Graphmuster, für welche ebenfalls Operatoren zu erzeugen und zu verknüpfen sind. Deshalb muss das Graphmuster GP einer SPARQL-Anfrage durch einen rekursiven Algorithmus modelliert werden.

Die Abbildung 4.1 auf Seite 42 zeigt eine Pseudocode-Darstellung dieses Algorithmus, welcher einen SQGM-Operator o für das übergebene Graphmuster $curGP$ erzeugt und zurückgibt. Die Darstellung gibt einen ersten Überblick über die rekursive Modellierung. Auf die einzelnen Schritte der Erzeugung von Operatoren für die unterschiedlichen Graphmuster wird im Folgenden genauer eingegangen. Die zweite Eingabe des Algorithmus ist ein graph-bereitstellender Operator, welcher die aktuelle Datenquelle repräsentiert. Diese wird benötigt, um die Datenquellen der erzeugten Operatoren für Graphmuster im Gültigkeitsbereich von RDF-Graph-Klauseln korrekt initialisieren zu können.

Das Graphmuster GP einer SPARQL-Anfrage wird letztendlich über den Aufruf

$$\textsc{CreateSQGMOperator}(GP, dflt)$$

generiert.

Einfache Graphmuster Für ein einfaches Graphmuster B (siehe Abschnitt 2.3.3 ab Seite 10) wird ein Graphmuster-Operator $o_B \in GPO$ mit

$$\begin{aligned}
o_B.\text{input} &:= gf \\
o_B.\text{output} &:= () \\
o_B.\text{provVars} &:= \begin{cases} V_B \cup \{\texttt{curDS}.\text{var}\} & \text{falls } \texttt{curDS} \in GSO \\ V_B & \text{sonst} \end{cases} \\
o_B.\text{triplePatterns} &:= TP_B \\
o_B.\text{constraints} &:= () \\
o_B.\text{contr} &:= \mathsf{F}
\end{aligned}$$

erzeugt, wobei gf ein neu zu erzeugender Graphdatenfluss ist ($gf \in GF$), TP_B die Liste der Tripelmuster in B und V_B die Menge der in TP_B erwähnten Variablen. Dann wird o_B dem SQGM hinzugefügt und über den neuen Graphdatenfluss gf mit der aktuellen Datenquelle verbunden.

$$\begin{aligned}
OP &:= OP \cup \{o_B\} \\
DF &:= DF \cup \{gf\} \\
\texttt{curDS}.\text{output} &:= \texttt{curDS}.\text{output} \circ gf
\end{aligned}$$

Wertebeschränkung Eine Wertebeschränkung C wird in der Syntax von SPARQL durch eine FILTER-Klausel ausgedrückt (siehe Abschnitt 2.3.6 ab Seite 12). Für C wird ebenfalls ein Graphmuster-Operator $o_C \in GPO$ mit den Eigenschaften

$$\begin{aligned}
o_C.\text{input} &:= gf \\
o_C.\text{output} &:= () \\
o_C.\text{provVars} &:= \begin{cases} V_C \cup \{\texttt{curDS}.\text{var}\} & \text{falls } \texttt{curDS} \in GSO \\ V_C & \text{sonst} \end{cases} \\
o_C.\text{triplePatterns} &:= () \\
o_C.\text{constraints} &:= (C) \\
o_C.\text{contr} &:= \mathsf{F}
\end{aligned}$$

```
ALGORITHMUS: CREATESQGMOPERATOR
EINGABE: curGP – das zu modellierende Graphmuster
EINGABE: curDS – aktuelle Datenquelle (ein graph-bereitstellender Operator)
AUSGABE: o – ein GP repräsentierender Operator

if ( curGP ist ein einfaches Graphmuster ) then
        Erzeuge o ∈ GPO für curGP;
        Verknüpfe curDS mit o;
elsif ( curGP ist eine Wertebeschränkung ) then
        Erzeuge o ∈ GPO für curGP;
        Verknüpfe curDS mit o;
elsif ( curGP = (P, O) ist ein optionales Graphmuster ) then
        o_P := CREATESQGMOPERATOR(P, curDS);
        o_O := CREATESQGMOPERATOR(O, curDS);
        Erzeuge o ∈ JO;
        Verknüpfe o_P und o_O mit o, wobei der Variablenfluss von o_O als optional gekennzeichnet wird;
elsif ( curGP = (P_1, ··· , P_n) ist eine Graphmuster-Gruppe ) then
        if ( n = 1 ) then
                o := CREATESQGMOPERATOR(P_1, curDS);
        else
                for ( 1 ≤ i ≤ n )
                        o_{P_i} := CREATESQGMOPERATOR(P_i, curDS);
                for ( 1 ≤ j < n )
                        Erzeuge o_j ∈ JO;
                        Verknüpfe o_{P_{j+1}} mit dem rechten Eingang von o_j;
                Verknüpfe o_{P_1} mit dem linken Eingang von o_1;
                if ( n > 2 ) then
                        for ( 2 ≤ j < n )
                                Verknüpfe o_{j-1} mit dem linken Eingang von o_j;
                o := o_{n-1};
elsif ( curGP = (P_1, ··· , P_n) ist eine Graphmuster-Vereinigung ) then
        for ( 1 ≤ i ≤ n )
                o_{P_i} := CREATESQGMOPERATOR(P_i, curDS);
        for ( 1 ≤ j < n )
                Erzeuge o_j ∈ UO;
                Verknüpfe o_{P_{j+1}} mit dem zweiten Eingang von o_j;
        Verknüpfe o_{P_1} mit dem ersten Eingang von o_1;
        if ( n > 2 ) then
                for ( 2 ≤ j < n )
                        Verknüpfe o_{j-1} mit dem ersten Eingang von o_j;
        o := o_{n-1};
elsif ( curGP = (g, P) ist eine RDF-Graph-Klausel ) then
        if ( g ist eine URI ) then
                if ( NG enthält einen Operator o_g für g ) then
                        ds := o_g;
                else
                        ERROR!
        else // g ist eine Variable
                if ( OP enthält einen Graph-Auswahloperator o_g für g ) then
                        ds := o_g;
                else
                        Erzeuge ds ∈ GSO;
                        Verknüpfe alle o_i ∈ NG mit ds;
        o := CREATESQGMOPERATOR(P, ds);
```

Abbildung 4.1: Rekursiver Algorithmus zur Erzeugung eines SQGM-Operators für ein Graphmuster

erzeugt, wobei gf ein neu zu erzeugender Graphdatenfluss ist ($gf \in GF$) und V_C die Menge der in C erwähnten Variablen. Der erzeugte Operator o_C wird dem SQGM hinzugefügt und über den neuen Graphdatenfluss gf mit der aktuellen Datenquelle verbunden.

$$\begin{aligned} OP &:= OP \cup \{o_C\} \\ DF &:= DF \cup \{gf\} \\ \texttt{curDS}.\text{output} &:= \texttt{curDS}.\text{output} \circ gf \end{aligned}$$

Es sei darauf hingewiesen, dass die Eigenschaft *contr*, welche angibt, ob die Wertebeschränkungen in einem Graphmuster-Operator nachweisbar widersprüchlich sind, bei der Konstruktion zunächst immer auf F gesetzt wird. Eine tatsächliche Untersuchung der Wertebeschränkungen und eine Feststellung von Widersprüchen kann in den späteren Schritten der Anfragekompilierung erfolgen.

Optionale Graphmuster Optionale Graphmuster (P, O) werden durch OPTIONAL-Klauseln in der SPARQL-Syntax repräsentiert (siehe Abschnitt 2.3.4 ab Seite 10). Für (P, O) werden zunächst rekursiv die Graphmuster P und O behandelt. Dabei entstehen die Operatoren o_P für P und o_O für O.

$$\begin{aligned} o_P &:= \textsc{CreateSQGMOperator}(P,\ \texttt{curDS}) \\ o_O &:= \textsc{CreateSQGMOperator}(O,\ \texttt{curDS}) \end{aligned}$$

Diese werden miteinander verbunden, indem ein Verbund-Operator $o_{(P,O)} \in JO$ mit den Eigenschaften

$$\begin{aligned} o_{(P,O)}.\text{input} &:= (vf_P, vf_O) \\ o_{(P,O)}.\text{output} &:= () \\ o_{(P,O)}.\text{provVars} &:= vf_P.\text{vars} \cup vf_O.\text{vars} \end{aligned}$$

erzeugt und dem SQGM hinzugefügt wird.

$$OP := OP \cup \{o_{(P,O)}\}$$

Die neu zu erstellenden Variablendatenflüsse $vf_P \in VF$ und $vf_O \in VF$ verbinden o_P und o_O mit $o_{(P,O)}$

$$\begin{aligned} o_P.\text{output} &:= o_P.\text{output} \circ vf_P \\ o_O.\text{output} &:= o_O.\text{output} \circ vf_O \end{aligned}$$

und haben die folgenden Eigenschaften.

$$\begin{aligned} vf_P.\text{vars} &:= o_P.\text{provVars} \\ vf_P.\text{optional} &:= \mathsf{F} \\ vf_O.\text{vars} &:= o_O.\text{provVars} \\ vf_O.\text{optional} &:= \mathsf{T} \end{aligned}$$

Auch sie werden dem SQGM hinzugefügt.

$$DF := DF \cup \{vf_P, vf_O\}$$

Graphmuster-Gruppen Eine Graphmuster-Gruppe $(P_1, \cdots, P_n)$ ist die Gruppierung mehrerer Graphmuster P_i (siehe Abschnitt 2.3.3 ab Seite 10). Bei der Modellierung werden zwei Fälle unterschieden. Im ersten Fall ist $n = 1$ und die Gruppe enthält nur ein Graphmuster P_1. Hier wird die Graphmuster-Gruppe nicht explizit modelliert, sondern durch den rekursiv für P_1 erzeugten Operator o_{P_1} repräsentiert.

$$o_{P_1} := \textsc{CreateSQGMOperator}(P_1,\ \texttt{curDS})$$

Im zweiten Fall ist $n > 1$ und die Graphmuster-Gruppe $(P_1, \cdots, P_n)$ kann als Verbund der Graphmuster P_i aufgefasst und durch Verbund-Operatoren modelliert werden. Deren Eigenschaft *input* wird mit einem Paar von neuen Variablendatenflüssen initialisiert, deren bereitstellende Operatoren den beiden verbundenen Graphmustern entsprechen. Werden durch eine Graphmuster-Gruppe mehr als zwei Graphmuster verbunden, gilt also $n > 2$, so müssen mehrere Verbund-Operatoren erstellt und diese untereinander verbunden werden.

Bei der Modellierung werden die P_i zunächst rekursiv behandelt. Dabei entstehen die Operatoren o_{P_1} bis o_{P_n}.

$$o_{P_i} := \textsc{CreateSQGMOperator}(P_i, \texttt{curDS})$$

Diese werden durch $n-1$ Verbund-Operatoren $jo_j \in JO$ miteinander verbunden. Die jo_j haben die Eigenschaften

$$\begin{aligned}
jo_1.\text{input} &:= \left(vf_{P_1}, vf_{P_2}\right) \\
\forall\, j \in \{2, \cdots n-1\} : jo_j.\text{input} &:= \left(vf_{j-1}, vf_{P_{j+1}}\right) \\
\forall\, j \in \{1, \cdots n-2\} : jo_j.\text{output} &:= (vf_j) \\
jo_{n-1}.\text{output} &:= () \\
jo_1.\text{provVars} &:= vf_{P_1}.\text{vars} \cup vf_{P_2}.\text{vars} \\
\forall\, j \in \{2, \cdots n-1\} : jo_j.\text{provVars} &:= vf_{j-1}.\text{vars} \cup vf_{P_{j+1}}.\text{vars}
\end{aligned}$$

und werden dem SQGM hinzugefügt.

$$OP := OP \cup \{jo_j \mid 1 \leq j < n\}$$

Die Konstruktion führt zu einem Verbund-Baum, dessen Form als *links-assoziativer Baum* (engl.: left-deep tree) bezeichnet wird.

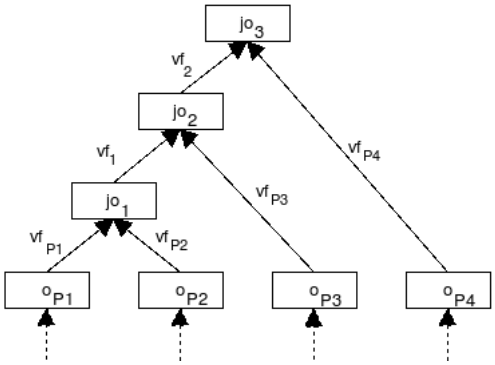


Abbildung 4.2: Ein links-assoziativer Verbund-Baum für eine Gruppe von vier Graphmustern

Bei diesem Mehrfach-Verbund, welcher in Abbildung 4.2 für $n = 4$ angedeutet ist, sind die Verbund-Operatoren über ihre linken Eingänge miteinander verknüpft und an jedem rechten Eingang hängt jeweils ein Operator der gruppierten Graphmuster. Demzufolge gibt es zwei Arten von Variablendatenflüssen, die während der Konstruktion zu erstellen sind. Die Datenflüsse $vf_1, \cdots, vf_{n-2} \in VF$ verketten die Verbund-Operatoren und über $vf_{P_1}, \cdots, vf_{P_n} \in VF$ sind die Operatoren der gruppierten Graphmuster mit jeweils einem der Verbund-Operatoren verknüpft.

$$\forall\, i \in \{1, \cdots, n\} : o_{P_i}.\text{output} := o_{P_i}.\text{output} \circ vf_{P_i}$$

Die Variablenflüsse haben die Eigenschaften

$$\begin{aligned}
\forall\, j \in \{1, \cdots, n-2\} : vf_j.\text{vars} &:= jo_j.\text{provVars} \\
\forall\, j \in \{1, \cdots, n-2\} : vf_j.\text{optional} &:= \mathsf{F} \\
\forall\, i \in \{1, \cdots, n\} : vf_{P_i}.\text{vars} &:= o_{P_i}.\text{provVars} \\
\forall\, i \in \{1, \cdots, n\} : vf_{P_i}.\text{optional} &:= \mathsf{F}
\end{aligned}$$

und werden dem SQGM hinzugefügt.

$$DF := DF \cup \{vf_j \mid 1 \leq j \leq n-2\} \cup \{vf_{P_i} \mid 1 \leq i \leq n\}$$

Der Operator jo_{n-1} ist die Wurzel des konstruierten Verbund-Baums und repräsentiert die Graphmuster-Gruppe in höheren Rekursionsebenen des Algorithmus zur Modellierung des Graphmusters einer SPARQL-Anfrage.

Es sei darauf hingewiesen, dass anstatt eines links-assoziative Verbund-Baums auch die Konstruktion eines alternativen, buschigen Baums möglich ist. Hier wurde der Einfachheit halber jedoch darauf verzichtet.

Beispiel 4.2 *Die Graphmuster-Gruppe der Anfrage aus Beispiel 4.1 besteht aus einem einfachen Graphmuster und einer Wertebeschränkung. Für diese werden zunächst rekursiv die Operatoren $o_{BGP} \in GPO$ mit den Eigenschaften*

$$\begin{aligned}
o_{BGP}.\text{input} &= d_{0_1} \\
o_{BGP}.\text{provVars} &= \{?m, ?r, ?h\} \\
o_{BGP}.\text{triplePatterns} &= \big((?m\ \texttt{exm:range}\ ?r), (?m\ \texttt{exm:height}\ ?h)\big) \\
o_{BGP}.\text{constraints} &= () \\
o_{BGP}.\text{contr} &= \mathsf{F}
\end{aligned}$$

und $o_C \in GPO$ mit

$$\begin{aligned}
o_C.\text{input} &= d_{0_2} \\
o_C.\text{provVars} &= \{?h\} \\
o_C.\text{triplePatterns} &= () \\
o_C.\text{constraints} &= \big((?h > 4000)\big) \\
o_C.\text{contr} &= \mathsf{F}
\end{aligned}$$

erzeugt, wobei d_{0_1} und d_{0_2} Graphdatenflüsse sind, welcher von dem in Beispiel 4.1 spezifizierten Operator dflt bereitgestellt werden, so dass

$$dflt.\text{output} = (d_{0_1}, d_{0_2})$$

gilt.
Die Gruppe selbst wird durch einen Verbund-Operator $o_1 \in JO$ mit den Eigenschaften

$$\begin{aligned}
o_1.\text{input} &= \big(d_{left}, d_{right}\big) \\
o_1.\text{provVars} &= \{?m, ?r, ?h\}
\end{aligned}$$

repräsentiert, wobei die Variablendatenflüsse d_{left} und d_{right} von den beiden rekursiv erzeugten Operatoren o_{BGP} und o_C bereitgestellt werden, so dass

$$\begin{aligned}
o_{BGP}.\text{output} &= (d_{left}) \\
o_C.\text{output} &= (d_{right})
\end{aligned}$$

gilt. □

Graphmuster-Vereinigungen Graphmuster-Vereinigungen $(P_1, \cdots, P_n)$ werden in der Syntax von SPARQL durch UNION-Klauseln ausgedrückt (siehe Abschnitt 2.3.5 ab Seite 11) und ihre Modellierung ist analog zur der von Graphmuster-Gruppen. Es werden zunächst rekursiv die vereinigten Graphmuster P_i behandelt, so dass die Operatoren o_{P_1} bis o_{P_n} entstehen.

$$o_{P_i} := \textsc{CreateSQGMOperator}(P_i, \texttt{curDS})$$

Diese werden durch $n-1$ Vereinigungs-Operatoren $uo_j \in UO$ miteinander vereinigt. Diese Vereinigungs-Operatoren haben die Eigenschaften

$$\begin{aligned}
uo_1.\text{input} &:= \left(vf_{P_1}, vf_{P_2}\right) \\
\forall\, j \in \{2, \cdots n-1\} : uo_j.\text{input} &:= \left(vf_{j-1}, vf_{P_{j+1}}\right) \\
\forall\, j \in \{1, \cdots n-2\} : uo_j.\text{output} &:= (vf_j) \\
uo_{n-1}.\text{output} &:= () \\
uo_1.\text{provVars} &:= vf_{P_1}.\text{vars} \cup vf_{P_2}.\text{vars} \\
\forall\, j \in \{2, \cdots n-1\} : uo_j.\text{provVars} &:= vf_{j-1}.\text{vars} \cup vf_{P_{j+1}}.\text{vars}
\end{aligned}$$

und werden dem SQGM hinzugefügt.

$$OP := OP \cup \{uo_j \mid 1 \leq j < n\}$$

Analog zur Konstruktionsvorschrift bei den Graphmuster-Gruppen entsteht auch hier ein linksassoziativer Baum und es gibt zwei Arten von zu erstellenden Variablendatenflüssen. Die Datenflüsse $vf_1, \cdots, vf_{n-2} \in VF$ verketten die Vereinigungs-Operatoren und über $vf_{P_1}, \cdots, vf_{P_n} \in VF$ sind die Operatoren der vereinigten Graphmuster mit jeweils einem der Vereinigungs-Operatoren verknüpft.

$$\forall\, i \in \{1, \cdots, n\} : o_{P_i}.\text{output} := o_{P_i}.\text{output} \circ vf_{P_i}$$

Die Variablenflüsse haben die Eigenschaften

$$\begin{aligned}
\forall\, j \in \{1, \cdots, n-2\} : vf_j.\text{vars} &:= uo_j.\text{provVars} \\
\forall\, j \in \{1, \cdots, n-2\} : vf_j.\text{optional} &:= \mathsf{F} \\
\forall\, i \in \{1, \cdots, n\} : vf_{P_i}.\text{vars} &:= o_{P_i}.\text{provVars} \\
\forall\, i \in \{1, \cdots, n\} : vf_{P_i}.\text{optional} &:= \mathsf{F}
\end{aligned}$$

und werden dem SQGM hinzugefügt.

$$DF := DF \cup \{vf_j \mid 1 \leq j \leq n-2\} \cup \{vf_{P_i} \mid 1 \leq i \leq n\}$$

Der konstruierte Vereinigungsbaum hat den Operator uo_{n-1} als Wurzel. Dieser repräsentiert die Graphmuster-Vereinigung in höheren Rekursionsebenen des Algorithmus zur Modellierung des Graphmusters einer SPARQL-Anfrage.

RDF-Graph-Klauseln RDF-Graph-Klauseln (g, P) werden in der SPARQL-Syntax durch eine GRAPH-Klausel repräsentiert (siehe Abschnitt 2.3.7 ab Seite 12). Für (g, P) wird zunächst, wie im Folgenden beschrieben, ein graph-bereitstellender Operator als aktuelle Datenquelle `ds` in Abhängigkeit von g festgelegt und dann wird rekursiv für das Graphmuster P der Operator o_P erzeugt.

$$o_P := \textsc{CreateSQGMOperator}(P, \texttt{ds})$$

Die erwähnte Festlegung der Datenquelle `ds` während der Behandlung von P ist abhängig von g. Einerseits kann g eine URI sein. Dann wird `ds` der Graphoperator $o \in NG$ mit $o.\text{uri} = g$ zugewiesen. Sollte das nicht möglich sein, weil NG keinen solchen Operator enthält, dann ist die SPARQL-Anfrage fehlerhaft. Andererseits kann g eine Variable sein. In diesem Fall wird OP nach

einem Graphauswahl-Operator $o \in GSO$ mit $o.\text{var} = g$ durchsucht. Existiert ein solcher Operator, so wird dieser `ds` zugewiesen. Existiert er nicht, so wird er mit den Eigenschaften

$$\begin{aligned} o.\text{input} &:= \big(gf_j \mid 1 \leq j \leq |NG|\big) \\ o.\text{output} &:= () \\ o.\text{var} &:= g \end{aligned}$$

konstruiert, wobei die gf_j zu erstellende Graphdatenflüsse sind, deren Konstruktion im Folgenden spezifiziert wird. Der neue Graphauswahl-Operator o wird dem SQGM hinzugefügt

$$OP := OP \cup \{o\}$$

und `ds` zugewiesen.

Da o die Auswahl aus der Menge der benannten RDF-Graphen repräsentiert, muss er mit allen Graph-Operatoren verbunden werden, welche diese benannten Graphen repräsentieren. Demzufolge werden die Graphdatenflüsse gf_j erzeugt

$$DF := DF \cup \big\{gf_j \mid 1 \leq j \leq |NG|\big\}$$

und jeweils einem der Graphoperatoren $o_j \in NG$ aus $NG = \big\{o_1, \cdots, |NG|\big\}$ als Ausgabedatenfluss zugeordnet.

$$\forall\, j \in \big\{1, \cdots, |NG|\big\} : o_j.\text{output} := o_j.\text{output} \circ gf_j$$

4.1.3 Modellierung von Lösungsanpassungen

Die Lösungsmenge einer Anfrage kann über Anpassungen der Lösungsfolge SM modifiziert werden. Diese werden in SPARQL durch die Klauseln DISTINCT, ORDER BY, LIMIT und OFFSET spezifiziert (siehe Abschnitt 2.3.8 ab Seite 14). Wenn eine Anfrage mindestens eine dieser Angaben zur Lösungsanpassung enthält, dann wird ein Lösungsanpassungs-Operator $o_{SM} \in SMO$ mit

$$\begin{aligned}
o_{SM}.\text{input} &:= \begin{cases} \text{undef.} & \text{falls } GP = \text{undef.} \\ vf & \text{sonst} \end{cases} \\
o_{SM}.\text{output} &:= () \\
o_{SM}.\text{provVars} &:= \begin{cases} \emptyset & \text{falls } GP = \text{undef.} \\ vf.\text{var} & \text{sonst} \end{cases} \\
o_R.\text{distinct} &:= \begin{cases} \mathsf{T} & \text{falls SELECT-Klausel mit Schlüsselwort DISTINCT} \\ \mathsf{F} & \text{sonst} \end{cases} \\
o_{SM}.\text{orderBy} &:= \begin{cases} ob & \text{falls } ob \in SM \wedge ob \text{ ist eine Liste von Sortierbedingung} \\ \text{undef.} & \text{sonst} \end{cases} \\
o_{SM}.\text{limit} &:= \begin{cases} l & \text{falls } l \in SM \wedge l \text{ ist eine Kardinalitätsbeschränkung} \\ \text{undef.} & \text{sonst} \end{cases} \\
o_{SM}.\text{offset} &:= \begin{cases} os & \text{falls } os \in SM \wedge os \text{ ist eine Offset-Angabe} \\ \text{undef.} & \text{sonst} \end{cases}
\end{aligned}$$

erzeugt, wobei vf ein neu zu erzeugender Variablendatenfluss ist und GP das Graphmuster der SPARQL-Anfrage. o_{SM} wird dem SQGM hinzugefügt.

$$OP := OP \cup \{o_{SM}\}$$

Wenn die SPARQL-Anfrage ein Graphmuster GP enthält, dann wurde dafür ein Baum von Operatoren erzeugt (siehe Abschnitt 4.1.2 ab Seite 41). Dessen Wurzel o_{GP} repräsentiert GP direkt und wird über einen neu zu erstellenden Variablendatenfluss $vf \in VF$ mit o_{SM} verbunden

$$o_{GP}.\text{output} := o_{GP}.\text{output} \circ vf$$

wobei vf die Eigenschaften

$$\begin{aligned} vf.\text{vars} &:= o_{GP}.\text{provVars} \\ vf.\text{optional} &:= \mathsf{F} \end{aligned}$$

hat und ebenfalls dem SQGM hinzugefügt wird.

$$DF := DF \cup \{vf\}$$

Wenn die repräsentierte Anfrage keine Klauseln zur Anpassung der Lösungsmenge enthält, so muss auch kein Lösungsanpassungs-Operator erstellt werden. Ist andererseits nur ein Teil der möglichen Klauseln in der Anfrage enthalten, so ist, wie in der Konstruktionsvorschrift zu erkennen, jeweils nur der Wert der, den vorhandenen Klauseln entsprechenden Eigenschaften definiert.

Beispiel 4.3 *Die Angaben zur Lösungsmengenanpassung der SPARQL-Anfrage aus Beispiel 4.1 (Seite 40) werden in einem SQGM durch einen Operator $o_2 \in SMO$ mit den Eigenschaften*

$$\begin{aligned} o_2.\text{input} &= d_1 \\ o_2.\text{provVars} &= \{?m, ?r, ?h\} \\ o_2.\text{distinct} &= \mathsf{T} \\ o_2.\text{orderBy} &= \big(\texttt{DESC}(?r)\big) \\ o_2.\text{limit} &= \textit{undef.} \\ o_2.\text{offset} &= \textit{undef.} \end{aligned}$$

repräsentiert, wobei d_1 ein Variablendatenfluss ist, welcher von dem in Beispiel 4.2 spezifizierten Operator o_1 bereitgestellt wird, so dass

$$o_1.\text{output} = (d_1)$$

gilt. □

4.1.4 Modellierung der Ergebnisform

Die Ergebnisform R einer SPARQL-Anfrage wird über eine SELECT-, ASK-, CONSTRUCT- oder DESCRIBE-Klausel bestimmt (siehe Abschnitt 2.3.2 ab Seite 9). In Abhängigkeit der tatsächlichen Klausel wird, wie im Folgenden beschrieben, ein ergebnis-erzeugender Operator o_R erstellt und dem SQGM als Operator, welcher die Ergebniserzeugung repräsentiert, hinzugefügt.

$$\begin{aligned} OP &:= OP \cup \{o_R\} \\ r &:= o_R \end{aligned}$$

Wenn die SPARQL-Anfrage Angaben zur Lösungsanpassung SM enthält, dann wurde dafür ein Lösungsanpassungs-Operator o_{SM} erzeugt (siehe Abschnitt 4.1.3 ab Seite 47). Dieser wird über einen neu zu erstellenden Variablendatenfluss $vf_{SM} \in VF$ mit o_R verbunden

$$o_{SM}.\text{output} := o_{SM}.\text{output} \circ vf : SM$$

wobei vf_{SM} die Eigenschaften

$$\begin{aligned} vf_{SM}.\text{vars} &:= o_{SM}.\text{provVars} \\ vf_{SM}.\text{optional} &:= \mathsf{F} \end{aligned}$$

hat und ebenfalls dem SQGM hinzugefügt wird.

$$DF := DF \cup \{vf_{SM}\}$$

Es könnte sein, dass die Anfrage keine Angaben zur Lösungsanpassung enthält und demzufolge auch kein Lösungsanpassungs-Operator erstellt wurde. Sie kann natürlich trotzdem ein Graphmuster GP enthalten. Ist das der Fall, dann wurde dafür eine Baum von Operatoren erzeugt (siehe Abschnitt 4.1.2 ab Seite 41). Sei der Operator o_{GP} die Wurzel dieses Baumes. Er repräsentiert GP direkt und wird über einen neu zu erstellenden Variablendatenfluss $vf_{GP} \in VF$ mit o_R verbunden

$$o_{GP}.\text{output} := o_{GP}.\text{output} \circ vf_{GP}$$

wobei vf_{GP} die Eigenschaften

$$\begin{aligned} vf_{GP}.\text{vars} &:= o_{GP}.\text{provVars} \\ vf_{GP}.\text{optional} &:= \mathsf{F} \end{aligned}$$

hat und dem SQGM hinzugefügt wird.

$$DF := DF \cup \{vf_{GP}\}$$

Der Eingabedatenfluss des neuen ergebnis-erzeugenden Operators o_R kann also von einem Lösungsanpassungs-Operator bereitgestellt werden, wenn die Anfrage Angaben zur Lösungsanpassung enthält. Ist das nicht der Fall, dann könnte er von einem, das Graphmuster repräsentierenden, Operator bereitgestellt werden. Beinhaltet die Anfrage auch kein Graphmuster, so hat o_R keinen Eingabedatenfluss.

Der Typ des neu zu erstellenden Operators o_R, und dessen Eigenschaften hängen von der tatsächlichen Ergebnisform ab. Bei einer SELECT-Klausel muss ein SELECT-Ergebnisoperator $o_R \in SRO$ mit den Eigenschaften

$$\begin{aligned} o_R.\text{input} &:= \begin{cases} vf_{SM} & \text{falls } SM \neq \text{undef.} \\ vf_{GP} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \\ \text{undef.} & \text{sonst} \end{cases} \\ o_R.\text{output} &:= () \\ o_R.\text{provVars} &:= \begin{cases} V_{\text{SELECT}} & \text{falls SELECT-Klausel ohne Symbol *} \\ vf_{SM}.\text{var} & \text{falls } SM \neq \text{undef.} \wedge \text{SELECT-Klausel mit Symbol *} \\ vf_{GP}.\text{var} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \wedge \text{SELECT mit Symbol *} \\ \emptyset & \text{sonst} \end{cases} \end{aligned}$$

erstellt werden. Hier sei besonders auf die Eigenschaft *provVars* hingewiesen. Werden in der repräsentierten SELECT-Klausel die für die Resultate relevanten Variablen V_{SELECT} aufgezählt, dann werden diese auch in *provVars* übernommen. Wenn die SELECT-Klausel aber das Symbol * enthält, also die Resultate aus allen in der Anfrage vorkommenden Variablen bestehen sollen, wird *provVars* durch Kopie der Menge der Variablen initialisiert, für die über den Eingabedatenfluss Bindungen übertragen werden.

Bei einer ASK-Klausel muss ein ASK-Ergebnisoperator $o_R \in ARO$ mit der Eigenschaft

$$o_R.\text{input} := \begin{cases} vf_{SM} & \text{falls } SM \neq \text{undef.} \\ vf_{GP} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \\ \text{undef.} & \text{sonst} \end{cases}$$

erzeugt werden.

Enthält die Anfrage eine CONSTRUCT-Klausel, so muss ein CONSTRUCT-Ergebnisoperator $o_R \in CRO$ mit den Eigenschaften

$$o_R.\text{input} := \begin{cases} vf_{SM} & \text{falls } SM \neq \text{undef.} \\ vf_{GP} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \\ \text{undef.} & \text{sonst} \end{cases}$$

$$o_R.\text{output} := ()$$

$$o_R.\text{template} := t$$

erzeugt werden, wobei t die in der CONSTRUCT-Klausel angegebene Konstruktionsschablone ist.

Bei einer DESCRIBE-Klausel muss ein DESCRIBE-Ergebnisoperator $o_R \in DRO$ mit den Eigenschaften

$$o_R.\text{input} := \begin{cases} vf_{SM} & \text{falls } SM \neq \text{undef.} \\ vf_{GP} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \\ \text{undef.} & \text{sonst} \end{cases}$$

$$o_R.\text{output} := ()$$

$$o_R.\text{describedResources} := \begin{cases} N_{\text{DESCRIBE}} & \text{falls DESCRIBE-Klausel ohne Symbol *} \\ vf_{SM}.\text{var} & \text{falls } SM \neq \text{undef.} \wedge \text{DESCRIBE-Klausel mit Symbol *} \\ vf_{GP}.\text{var} & \text{falls } SM = \text{undef.} \wedge GP \neq \text{undef.} \wedge \text{DESCRIBE mit *} \\ \emptyset & \text{sonst} \end{cases}$$

erstellt werden, wobei N_{DESCRIBE} die Menge der nach dem Schlüsselwort `DESCRIBE` aufgezählten Variablen und URIs ist.

Beispiel 4.4 *Die Ergebniserzeugung der SPARQL-Anfrage aus Beispiel 4.1 (Seite 40) wird in einem SQGM durch einen Operator $o_3 \in SRO$ mit den Eigenschaften*

$$o_3.\text{input} = d_2$$
$$o_3.\text{provVars} = \{?r\}$$

repräsentiert, wobei d_2 ein Variablendatenfluss ist, welcher von dem in Beispiel 4.3 spezifizierten Operator o_2 bereitgestellt wird, so dass

$$o_2.\text{output} = (d_2)$$

gilt. Die Abbildung 4.3 zeigt das erzeugte SQGM. □

4.1.5 Eigenschaften des modellierten SQGM

Das in diesem Abschnitt angegebene Vorgehen erzeugt ein SQGM, welches ein zyklenfreier, gerichteter und zusammenhängender Graph mit den Operatoren als Knoten und den Datenflüssen als Kanten ist. Dieser kann in zwei Ebenen unterteilt werden. Die *untere Ebene* ist der Teilgraph, welcher, bis auf die resultat-erzeugenden, alle graph-bereitstellenden Operatoren als Knoten beinhaltet. Dieser Teilgraph ist ein Wald, dessen Teile einzelne Graphoperatoren und gerichtete Bäume mit Graph-Verschmelzungs- und Graph-Auswahloperatoren sein können. Die *obere Ebene* ist der Teilgraph, welcher alle anderen Operatoren als Knoten beinhaltet. Hierbei handelt es sich um einen, zur Wurzel gerichteten Baum, dessen Blätter immer Graphmuster-Operatoren sind. Diese haben jeweils eine eingehende Kante, welche von einem graph-bereitstellenden Operator aus der unteren Ebene ausgeht und die Datenquelle des entsprechenden Graphmuster-Operator repräsentiert. Von einigen der graph-bereitstellenden Operatoren der unteren Ebene können so mehrere

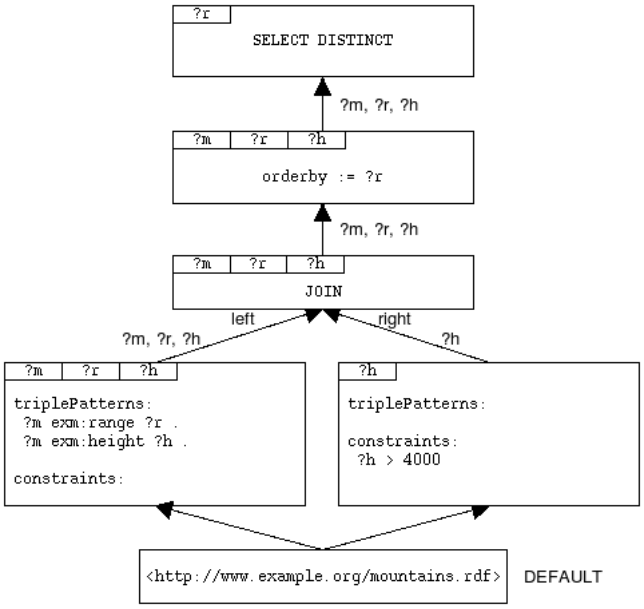


Abbildung 4.3: Grafische Darstellung des SQGM mit den Operatoren aus den Beispielen 4.1, 4.2, 4.3 und 4.4

Kanten zu verschiedenen Graphmuster-Operatoren der oberen Ebene ausgehen. Es gibt aber keine Kanten von der oberen in die untere Ebene. Demzufolge ist das gesamte SQGM frei von Zyklen.

Als weitere Eigenschaft der erzeugten Graphen, haben alle Operator-Knoten, bis auf die resultat-erzeugenden Operatoren, mindestens eine ausgehende Kante. Da der Graph zyklenfrei und zusammenhängend ist, hat der jeweils längste, von jedem Operator-Knoten ausgehende, Pfad immer den resultat-erzeugenden Operator als Endknoten. Demzufolge kann dieser Operator als *Wurzel* des SQGM bezeichnet werden.

Die obere Ebene eines SQGM kann ihrerseits wieder in 3 Schichten unterteilt werden. Die *Basis-Schicht* bilden die Blätter des Baumes, welche immer benachbarte Graphmuster-Operatoren sind. Diesen folgen möglicherweise Verbund- und Vereinigungs-Operatoren, welche in der *Zusammenführungsschicht* liegen. An der Spitze befindet sich die *Kopf-Schicht*, welche einen eventuell vorhandener Lösungsanpassungs-Operator und die Wurzel des SQGM beinhaltet. Diese Unterteilung ist eine Folge der Eigenschaften des SQGM und der Konstruktionsvorschrift. Damit steht auch fest, dass der Pfad von einem Graphmuster-Operator zur Wurzel keinen weiteren Graphmuster-Operator beinhalten kann.

Beispiel 4.5 *In der Abbildung 4.4 sind die Ebenen und Schichten des SQGM aus dem Beispiel 4.4 dargestellt.* □

4.2 Normalisierung

Um die Optimierungen einer SPARQL-Anfrage zu vereinfachen, muss das initiale SQGM, welches nach dem im vorangegangenen Abschnitt spezifizierten Vorgehen aus der Anfrage erzeugt wurde, in eine standardisierte Form gebracht werden. In diesem Abschnitt wird zunächst eine entsprechende Normalform für das SQGM vorgestellt. Danach werden deren einzelne Eigenschaften diskutiert und gegebenenfalls entsprechende Transformationsregeln spezifiziert. Zum Abschluss wird sie mit anderen Normalformen verglichen.

Bei einer algebraischen Darstellung werden algebraische Ausdrücke, die eine bestimmte Form haben, als normalisiert definiert. Die Normalform eines SQGM kann über die Festlegung bestimm-

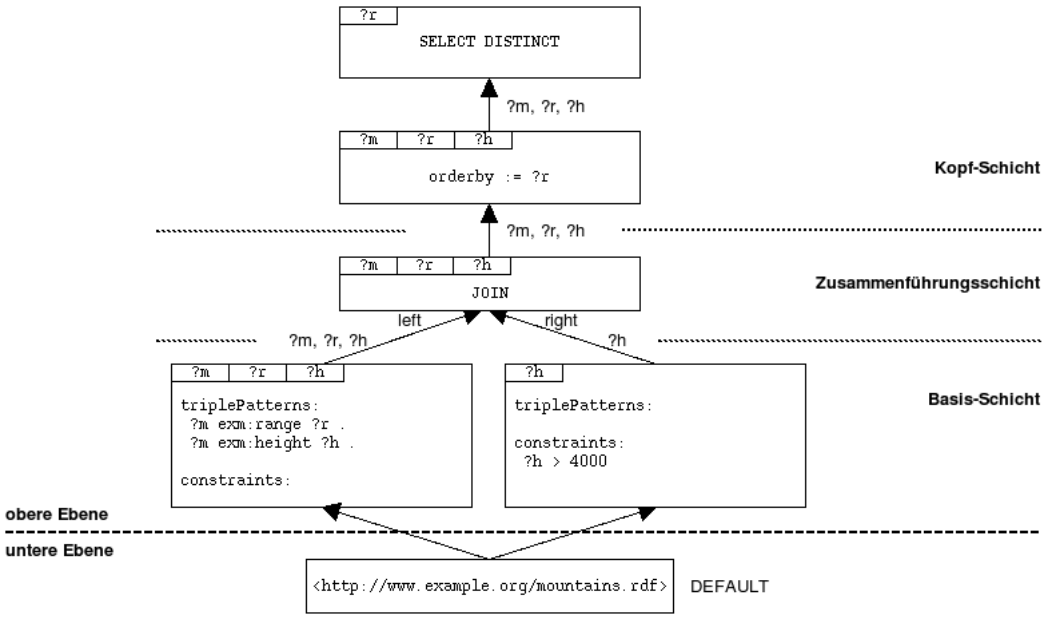


Abbildung 4.4: Ebenen und Schichten eines SQGM

ter Eigenschaften spezifiziert werden, welche ein normalisiertes SQGM haben muss. Die folgende Definition zählt diese Eigenschaften auf. Im Anschluss werden diese einzeln diskutiert.

Definition 4.1 Ein *normalisiertes SQGM* ist ein SQGM $(OP, DF, r, dflt, NG)$, welches die folgenden Eigenschaften hat.

1. Eingeschränkte Ausgänge – Jeder variablen-bereitstellende Operator aus OP stellt genau einen Variablendatenfluss bereit.

$$\forall\, o \in VPOP \subset OP : \, |\, o.\text{output}\, | = 1$$

2. Lokalität von Variablenbindungen – Jeder variablen-bereitstellende Operator aus OP stellt nur für die Variablen Bindungen bereit, für welche er Bindungen erzeugt oder deren Bindungen er oder ein im SQGM nachfolgender Operator benutzt. Weiterhin werden über jeden Variablendatenfluss nur die Variablenbindungen übergeben, für deren Variablen der benutzende Operator wiederum Bindungen bereitstellt.

3. Normalisierte Wertebeschränkungen – Keine der Wertebeschränkungen aller Graphmuster-Operatoren aus OP darf ein logisches Und enthalten.

4. Freiheit von leeren Graphmuster-Operatoren – Keiner der Graphmuster-Operatoren aus OP darf eine leere Menge an Tripelmustern und eine leere Menge an Wertebeschränkungen haben.

$$\forall\, o \in GPO \subset OP : \Big[\Big(o.\text{triplePatterns} \neq ()\Big) \vee \Big(o.\text{constraints} \neq ()\Big)\Big]$$

□

4.2.1 Eingeschränkte Ausgänge

Die erste Eigenschaft, welche die Kardinalität der Ausgänge variablen-bereitstellender Operatoren beschränkt, wird durch das spezifizierte Vorgehen zur Konstruktion eines initialen SQGM aus einer SPARQL-Anfrage (siehe Abschnitt 4.1 ab Seite 38) garantiert. Wegen dieser Eigenschaft ist der

Teilgraph eines SQGM, der nur die variablen-bereitstellenden Operatoren beinhaltet, ein Baum. Diese Eigenschaft wird insbesondere bei der Restrukturierung von Verbund- und Vereinigungsbäumen innerhalb des SQGM ausgenutzt (siehe beispielsweise Abschnitt 6.3.2 ab Seite 93), da sie die entsprechenden Transformationsregeln wesentlich vereinfacht.

4.2.2 Lokalität von Variablenbindungen

Für die zweite Eigenschaft, die Lokalität von Variablenbindungen, gibt es den folgenden Grund. In SPARQL sind Variablen global gültig. Trotzdem ist es ist möglich, dass sie nur in Teilbereichen einer Anfrage, zum Beispiel nur innerhalb einer RDF-Graph-Klausel oder nur innerhalb genau eines von mehreren vereinigten Graphmustern, verwendet werden. Demzufolge können diese Variablen als lokal bezüglich dieses Graphmusters betrachtet werden. Durch die Eigenschaft wird verlangt, dass die Lokalität von Variablenbindungen in einem SQGM ersichtlich ist, indem eine Variable zwar von einem Operator bereitgestellt, aber in keinem, seiner ausgehenden Datenflüsse übergeben wird. Diese Kennzeichnung könnte unter anderem bei einer verteilten Anfrageausführung Vorteile bringen, da auf ihrer Basis lediglich eine Teilmenge aller Variablenbindungen für die Übertragung festlegt werden muss.

Eine solche Beschränkung der bereitgestellten und übermittelten Variablenbindungen kann nicht bereits während der Konstruktion des SQGM vorgenommen werden, denn die mögliche Nichtverwendung von Variablen muss in der Operatorhierarchie nach unten propagiert werden. Dafür muss bereits ein initialer SQGM existieren.

Die folgenden Regeln dienen zum Propagieren nicht benötigter Variablenbindungen, wobei zunächst für die resultat-erzeugenden Operatoren, als mögliche Wurzelknoten im SQGM, nicht verwendete Variablen gekennzeichnet werden.

Transformationsregel 4.1 – PropagateSROUnusedVar

Definitionsbereich: $o \in SRO \subset OP$

Voraussetzung: Es gelte:

i) $o.\text{input} \neq \text{undef.}$
ii) $\exists\, v \in o.\text{input.vars} : v \notin o.\text{provVars}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{input.vars} := o.\text{input.vars} \setminus \{v\}$

Erläuterung: Diese Regel betrachtet einen SELECT-Ergebnisoperator o und eine Variable v für die Bindungen über den Eingangsdatenfluss von o übermittelt werden. Wenn v nicht von o bereitgestellt wird, dann entfernt diese Regel die Variable v aus der Menge der Variablen, für die über den Eingangsdatenfluss Bindungen übermittelt werden.

Begründung: Diese Regel behält die Semantik des SQGM bei, weil v nicht selektiert wird und deswegen auch nicht für die Abarbeitung von o benötigt wird. □

Transformationsregel 4.2 – PropagateDROUnusedVar

Definitionsbereich: $o \in DRO \subset OP$

Voraussetzung: Es gelte:

i) $o.\text{input} \neq \text{undef.}$
ii) $\exists\, v \in o.\text{input.vars} : v \notin o.\text{describedResources}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{input.vars} := o.\text{input.vars} \setminus \{v\}$

Erläuterung: Diese Regel betrachtet einen DESCRIBE-Ergebnisoperator o und eine Variable v für die Bindungen über den Eingangsdatenfluss von o übermittelt werden. Wenn v nicht von o beschrieben wird, dann entfernt diese Regel die Variable v aus der Menge der Variablen, für die über den Eingangsdatenfluss Bindungen übermittelt werden.

Begründung: Diese Regel behält die Semantik des SQGM bei, weil v nicht durch o beschrieben wird und deswegen auch nicht für die Abarbeitung von o benötigt wird. □

Transformationsregel 4.3 – PropagateCROUnusedVar

Definitionsbereich: $o \in CRO \subset OP$

Voraussetzung: Es muss

i) $o.\text{input} \neq \text{undef.}$
ii) $\exists\, v \in o.\text{input.vars} : v \notin TV_o$

gelten, wobei TV_o die Menge der in o.template erwähnten Variablen ist.

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{input.vars} := o.\text{input.vars} \setminus \{v\}$

Erläuterung: Für den CONSTRUCT-Ergebnisoperator o entfernt diese Regel die Variable v aus der Menge der, über den Eingangsdatenfluss übermittelten Variablenbindungen, wenn v nicht in der Konstruktionsschablone von o enthalten ist.

Begründung: Diese Regel behält die Semantik des SQGM bei, da v nicht bei der Konstruktion verwendet wird und deswegen auch nicht für die Abarbeitung von o benötigt wird. □

Es sei darauf hingewiesen, dass jede der drei obigen Regeln mehrmals angewandt werden könnte. Das ist der Fall, wenn mehrere der, vom Eingangsdatenfluss bereitgestellten Variablenbindungen nicht innerhalb des entsprechenden resultat-erzeugenden Operators verwendet bzw. ausgewertet werden.

Die folgenden Regeln propagieren die Nichtverwendung von Variablenbindungen für Lösungsanpassungs-, Verbund- und Vereinigungsoperatoren.

Transformationsregel 4.4 – PropagateSMOUnusedVar

Definitionsbereich: $o \in SMO \subset OP$

Voraussetzung: Zunächst muss

i) $o.\text{input} \neq \text{undef.}$

gelten. Sei $v \in o.\text{input.vars}$. Es muss

ii) $\forall\, d \in o.\text{output} : v \notin d.\text{vars}$
iii) $v \notin O_o$

gelten, wobei O_o die Menge der in o.orderBy erwähnten Variablen ist.

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{provVars} := o.\text{provVars} \setminus \{v\}$
ii) $o.\text{input.vars} := o.\text{input.vars} \setminus \{v\}$

Erläuterung: Wenn die Variable v im Lösungsanpassungs-Operator o nicht zur Sortierung der Lösungsmenge benötigt und ihre Bindugen über keinen Ausgangsdatenfluss übermittelt werden, dann entfernt diese Regel v aus der Menge der bereitgestellten Variablen und aus der Menge der, über den Eingangsdatenfluss übermittelten Variablen.

Begründung: Diese Regel behält die Semantik des SQGM bei, da v weder zur Sortierung, noch von einem nachfolgenden Operator benötigt wird. □

Transformationsregel 4.5 – PropagateUnreqVarJoin

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$ und $v \in \Big(\big(i_{\text{left}}.\text{vars} \cup i_{\text{right}}.\text{vars}\big) \setminus \big(i_{\text{left}}.\text{vars} \cap i_{\text{right}}.\text{vars}\big)\Big)$. Es gelte:

i) $\forall\, d \in o.\text{output} : v \notin d.\text{vars}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{provVars} := o.\text{provVars} \setminus \{v\}$
ii) $i_{\text{left}}.vars := i_{\text{left}}.vars \setminus \{v\}$
iii) $i_{\text{right}}.vars := i_{\text{right}}.vars \setminus \{v\}$

Erläuterung: Wenn die Bindungen für die Variable v im Verbund-Operator o nur vom linken oder nur vom rechten Eingangsdatenfluss zur Verfügung gestellt und über keinen Ausgangsdatenfluss übermittelt werden, dann entfernt diese Regel v aus der Menge der bereitgestellten Variablen und aus der Menge der, über die Eingangsdatenflüsse übermittelten Variablen.

Begründung: Da die Bindungen für v nur von einem Eingangsdatenfluss bereitgestellt werden, sind sie für die Berechnung der Verbund-Operation nicht notwendig. Weiterhin werden sie von keinem nachfolgenden Operator benötigt. Deshalb behält diese Regel die Semantik des SQGM bei. □

Transformationsregel 4.6 – PropagateUnreqVarUnion

Definitionsbereich: $o \in UO \subset OP$

Voraussetzung: Seien $(i_1, i_2) := o.\text{input}$ und $v \in (i_1.\text{vars} \cup i_2.\text{vars})$. Es gelte:

i) $\forall\, d \in o.\text{output} : v \notin d.\text{vars}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{provVars} := o.\text{provVars} \setminus \{v\}$
ii) $i_1.vars := i_1.vars \setminus \{v\}$
iii) $i_2.vars := i_2.vars \setminus \{v\}$

Erläuterung: Wenn im Vereinigungs-Operator o die, von einem oder beiden Eingangsdatenflüsse zur Verfügung gestellten Bindungen für die Variable v über keinen Ausgangsdatenfluss übermittelt werden, dann entfernt diese Regel v aus der Menge der bereitgestellten Variablen und aus der Mengen der, über beide Eingangsdatenflüsse übermittelten Variablen.

Begründung: Diese Regel behält die Semantik des SQGM bei, da die Bindungen für v von keinem nachfolgenden Operator benötigt werden. □

Auch hier sei darauf hingewiesen, dass jede der angegebenen Regeln mehrmals angewandt werden könnte.

Es kann festgestellt werden, dass die Regeln in ihrer Anwendbarkeit voneinander abhängig sind. Die Anwendung einer der Transformationsregeln erfüllt unter Umständen die Voraussetzungen für weitere Regeln. Wenn beispielsweise die Regel PropagateSROUnusedVar für einen SELECT-Ergebnisoperator o ausgeführt werden konnte, dann ist es möglich, dass die Regel PropagateSMOUnusedVar für einen im SQGM direkt unterhalb von o befindlichen Lösungsanpassungs-Operator ausführbar wird. Andererseits könnte PropagateSROUnusedVar auch noch einmal bezüglich einer anderen Variablenbindung für o ausgeführt werden können.

In der Tabelle 4.1 auf Seite 57 sind alle Abhängigkeiten für die spezifizierten Regeln aufgelistet. Diese Tabelle ordnet jeder ausgeführten Regel mehrere mögliche Folgeregeln für jeweils einen Operator zu. Eine Folgeregel kommt jedoch nur in Frage, wenn die jeweils angegebene Bedingung erfüllt ist. In diesem Fall könnten die Voraussetzungen der möglichen Folgeregel durch die Anwendung der ursprünglichen Regel erfüllt worden sein. Die Angabe der Bedingung und des zu verwendenden Operators bezieht sich auf Objekte, welche im Rahmen der ausgeführten Regel definiert wurden.

Die rekursive Bestimmung von Folgeregeln und damit auch die vollständige und erschöpfende Anwendung aller Regeln und möglicher Folgeregeln ist aus dem folgenden Grund deterministisch. Jede der Regeln hat als mögliche Folgeregel entweder sich selbst bezüglich des aktuellen Operators oder irgendeine der angegebenen Regeln bezüglich eines, im SQGM unterhalb des aktuellen Operators befindlichen Operators. Im ersten Fall betrachten die Regeln jeweils eine endlich große Menge von Variablenbindungen und schränken diese bei ihrer Ausführung ein. Hier setzt die Endlichkeit der Variablenbindungen der Möglichkeit zur Ausführung eine Grenze. Im zweiten Fall wird die Grenze durch die Endlichkeit der Größe des SQGM gesetzt.

Die vollständige und erschöpfende Anwendung aller angegebenen Regeln und möglichen Folgeregeln führt zu einem SQGM, welches die zweite Eigenschaft der Normalform, die Lokalität von Variablenbindungen, erfüllt.

Beispiel 4.6 *Wenn man auf das in Abbildung 4.3 (Seite 51) dargestellte SQGM die Transformationsregeln zum Propagieren nicht benötigter Variablenbindungen erschöpfend anwendet, erhält man das in der Abbildung 4.5 dargestellte, normalisierte SQGM.*

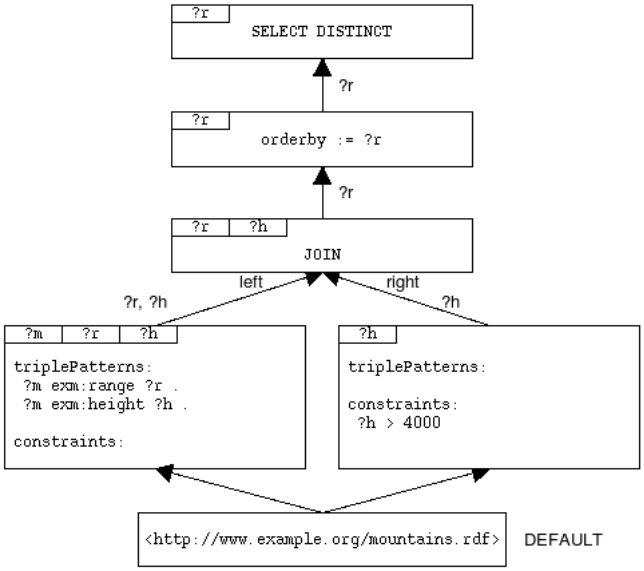


Abbildung 4.5: Normalisierte Form des SQGM aus Abbildung 4.3 □

4.2.3 Normalisierte Wertebeschränkungen

Mit der dritten Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) wird gefordert, dass keiner der logischen Ausdrücke der Wertebeschränkungen eines der Graphmuster-Operatoren ein logisches Und enthalten darf. Diese Konjunktionen sind nicht notwendig. Die SPARQL-Spezifikation schreibt vor, dass bei einer Gruppe von Wertebeschränkungen jede einzelne erfüllt sein muss und verwendet somit implizit die Semantik der Konjunktion. Demzufolge können Ausdrücke mit logischem Und in eine konjunktive Normalform umgeformt und dann in mehrere Wertebeschränkungen aufgeteilt werden.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verwendender Operator
PropagateSROUnusedVar	$o.\text{input.vars} \neq \emptyset$	PropagateSROUnusedVar	o
	$\mathsf{ProvOp}(o.\text{input}) \in SMO$	PropagateSMOUnusedVar	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(o.\text{input})$
PropagateDROUnusedVar	$o.\text{input.vars} \neq \emptyset$	PropagateDROUnusedVar	o
	$\mathsf{ProvOp}(o.\text{input}) \in SMO$	PropagateSMOUnusedVar	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(o.\text{input})$
PropagateCROUnusedVar	$o.\text{input.vars} \neq \emptyset$	PropagateCROUnusedVar	o
	$\mathsf{ProvOp}(o.\text{input}) \in SMO$	PropagateSMOUnusedVar	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(o.\text{input})$
PropagateSMOUnusedVar	$o.\text{input.vars} \neq \emptyset$	PropagateSMOUnusedVar	o
	$\mathsf{ProvOp}(o.\text{input}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(o.\text{input})$
	$\mathsf{ProvOp}(o.\text{input}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(o.\text{input})$
PropagateUnreqVarJoin	$o.\text{input.vars} \neq \emptyset$	PropagateUnreqVarJoin	o
	$\mathsf{ProvOp}(i_{\text{left}}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(i_{\text{left}})$
	$\mathsf{ProvOp}(i_{\text{right}}) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(i_{\text{right}})$
	$\mathsf{ProvOp}(i_{\text{left}}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(i_{\text{left}})$
	$\mathsf{ProvOp}(i_{\text{right}}) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(i_{\text{right}})$
PropagateUnreqVarUnion	$o.\text{input.vars} \neq \emptyset$	PropagateUnreqVarUnion	o
	$\mathsf{ProvOp}(i_1) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(i_1)$
	$\mathsf{ProvOp}(i_2) \in JO$	PropagateUnreqVarJoin	$\mathsf{ProvOp}(i_2)$
	$\mathsf{ProvOp}(i_1) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(i_1)$
	$\mathsf{ProvOp}(i_2) \in UO$	PropagateUnreqVarUnion	$\mathsf{ProvOp}(i_2)$

Tabelle 4.1: Abhängigkeiten zwischen den Transformationsregeln zum Propagieren nicht benötigter Variablenbindungen

Der Vorteil einer solchen Aufteilung besteht darin, dass die kleineren Ausdrücke eventuell, jeder für sich, eine kleinere Menge an Variablen beschränken und sich damit mehr Möglichkeiten für spätere Optimierungen ergeben. Bei der Umwandlung des SQGM in einen Ausführungsplan können die einzelnen Wertebschränkungen natürlich problemlos wieder zusammengefasst werden, wenn das bei einem bestimmten Speicher- und Zugriffsmodell vorteilhaft ist.

Die folgenden beiden Regeln dienen zur Umwandlung der logischen Ausdrücke von Wertebeschränkungen in konjunktive Normalformen und zur Aufspaltung von Wertebeschränkungen deren Ausdrücke Konjunktionen sind.

Transformationsregel 4.7 – NormalizeConstraint

Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Sei $c \in o.\text{constraints}$. Es gelte:

i) der Ausdruck von c enthält ein logisches Und

ii) der Ausdruck von c ist keine konjunktive Normalform

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ durch die Transformation

i) $o.\text{constraints} := \big(c^* \mid c^* \in o.\text{constraints} \wedge c^* \neq c\big) \circ \big(c'\big)$

wobei c' die konjunktive Normalform zu c ist.

Erläuterung: Wenn der Graphmuster-Operator o eine Wertebeschränkung c enthält, deren logischer Ausdruck ein logisches Und enthält, dann wird der logische Ausdruck dieser Wertebeschränkung in eine konjunktive Normalform gebracht.

Begründung: Diese Regel hat keinen Einfluss auf das Ergebnis der Anfrage, da c' semantisch äquivalent zu c ist. □

Transformationsregel 4.8 – SplitConstraint

Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Sei $c \in o.\text{constraints}$. Es gelte:

i) $c = c_1 \wedge c_2$ *(c ist eine Konjunktion)*

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{constraints} := \big(c^* \mid c^* \in o.\text{constraints} \wedge c^* \neq c\big) \circ \big(c_1, c_2\big)$

Erläuterung: Wenn der Graphmuster-Operator o eine Wertebeschränkung c enthält, deren logischer Ausdruck eine Konjunktion zweier logischer Ausdrücke c_1 und c_2 ist, dann wird c in o durch c_1 und c_2 ersetzt.

Begründung: Diese Regel behält die Semantik des SQGM bei, da bei einer Gruppe von Wertebeschränkungen jede einzelne erfüllt sein muss und somit implizit die Semantik der Konjunktion verwendet wird. □

Die Tabelle 4.2 stellt die Abhängigkeiten der beiden Regeln in Bezug auf mögliche Folgeregeln dar. Ihre Anwendbarkeit ist begrenzt, da ein SQGM nur eine endliche Anzahl von Graphmuster-Operatoren enthält und jeder Graphmuster-Operator nur eine endlich große Menge von Wertebeschränkungen. Eine vollständige und erschöpfende Anwendung dieser Regeln führt zu einem SQGM, welches die dritte Eigenschaft der Normalform erfüllt.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verwendender Operator
NormalizeConstraint	o.constraints $\neq$ ()	NormalizeConstraint	o
	o.constraints $\neq$ ()	SplitConstraint	o
SplitConstraint	–	SplitConstraint	o

Tabelle 4.2: Abhängigkeiten zwischen den Regeln zur Normalisierung von Wertebeschränkungen

4.2.4 Freiheit von leeren Graphmuster-Operatoren

Die vierte Eigenschaft fordert, dass ein normalisiertes SQGM keine leeren Graphmuster-Operatoren enthält. Als leer werden Graphmuster-Operatoren verstanden, wenn die Listen der Tripelmuster und Wertebeschränkungen, repräsentiert durch die Eigenschaften *triplePatterns* und *constraints*, gleichzeitig leer sind. Da solche Operatoren keinen Beitrag zur Lösungsmenge einer Graphmusterübereinstimmung liefern, vergrößern sie ein SQGM nur unnötig und sollten entfernt werden.

Leere Graphmuster-Operatoren können bei der Konstruktion eines SQGM entstehen, weil in SPARQL leere Teile von Graphmustern syntaktisch möglich sind. Diese könnten unter anderem bei einer automatischen Anfragegenerierung erzeugt worden sein.

Die folgenden Regeln dienen zur Entfernung von leeren Graphmuster-Operatoren. Zusätzlich zu den entsprechenden Operatoren werden durch die Regeln auch betroffene Datenflüsse aus dem SQGM entfernt. Es sei darauf hingewiesen, dass bei einer Implementierung darauf zu achten ist, diese Datenflüsse vor dem Löschen aus den Eigenschaften *input* und *output* der, durch sie verbundenen Operatoren zu entfernen.

Transformationsregel 4.9 – RemoveEmptyInput

Definitionsbereich: $o \in (SRO \cup DRO \cup CRO \cup ARO \cup SMO) \subset OP$

Voraussetzung: Zunächst muss

i) $o.\text{input} \neq \text{undef}$.

gelten. Sei $o_{\text{empty}} := \mathsf{ProvOp}(o.\text{input})$. Es muss

ii) $o_{\text{empty}} \in GPO$
iii) $o_{\text{empty}}.\text{triplePatterns} = ()$
iv) $o_{\text{empty}}.\text{constraints} = ()$
v) $ds \notin GSO$

gelten, wobei $ds \in \mathsf{DS}(o_{\text{empty}})$.

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $DF' := DF \setminus \{o.\text{input}, o_{\text{empty}}.\text{input}\}$
ii) $OP' := OP \setminus \{o_{\text{empty}}\}$

Erläuterung: Wenn sich im SQGM direkt unterhalb des Operators o ein leerer Graphmuster-Operator o_{empty} befindet, so wird dieser und dessen Verbindung zu o aus dem SQGM entfernt.
Die einzige Ausnahme bilden leere Graphmuster-Operatoren, die RDF-Graph-Klauseln mit einer Variablen als Graphspezifikation repräsentieren. Deren Entfernung verhindert die Voraussetzung v).

Begründung: Der leere Graphmuster-Operator o_{empty} leistet keinen Beitrag zur Lösungsmenge und kann deshalb entfernt werden ohne das Ergebnis der Anfrage zu beeinflussen. □

Transformationsregel 4.10 – RemoveEmptyJoinInputLeft

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $o_{\text{left}} \in GPO$
ii) $o_{\text{left}}.\text{triplePatterns} = ()$
iii) $o_{\text{left}}.\text{constraints} = ()$
iv) $ds \in GSO \rightarrow \mathsf{DS}(o_{\text{left}}) \neq \mathsf{DS}(o)$ wobei $ds \in \mathsf{DS}(o_{\text{left}})$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_{\text{right}}.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_{\text{right}}.\text{optional}\big)$
iii) $o_{\text{right}}.\text{output} := \big(o_{\text{right}}.\text{output} \setminus \{i_{\text{right}}\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_{\text{left}}, i_{\text{right}}, o_{\text{left}}.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_{\text{left}}\}$

Erläuterung: Wenn für einen Verbund-Operator o der bereitstellende Operator o_{left} des linken Eingabedatenflusses i_{left} ein leerer Graphmuster-Operator ist, so wird der Verbund aus o, o_{left} und o_{right} durch o_{right} ersetzt. Demzufolge werden o und o_{left} und deren Verbindungen untereinander aus dem SQGM entfernt und die Eigenschaften weiterer betroffender Datenflüsse angepasst.
Die einzige Ausnahme bilden leere Graphmuster-Operatoren, die RDF-Graph-Klauseln mit einer Variablen als Graphspezifikation repräsentieren. Deren Entfernung verhindert die Voraussetzung iv).

Begründung: Der leere Graphmuster-Operator o_{left} liefert keine Variablenbindungen und muss deshalb nicht mit der Lösungsmenge des Operators o_{right} verbunden werden. Demzufolge hat die Entfernung von o und o_{left} keinen Einfluss auf das Ergebnis der Anfrage. □

Transformationsregel 4.11 – RemoveEmptyJoinInputRight

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $o_{\text{right}} \in GPO$
ii) $o_{\text{right}}.\text{triplePatterns} = ()$
iii) $o_{\text{right}}.\text{constraints} = ()$
iv) $ds \in GSO \rightarrow \mathsf{DS}(o_{\text{right}}) \neq \mathsf{DS}(o)$ wobei $ds \in \mathsf{DS}(o_{\text{right}})$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_{\text{left}}.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_{\text{left}}.\text{optional}\big)$
iii) $o_{\text{left}}.\text{output} := \big(o_{\text{left}}.\text{output} \setminus \{i_{\text{left}}\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_{\text{left}}, i_{\text{right}}, o_{\text{right}}.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_{\text{right}}\}$

Erläuterung: Diese Regel ist das Pendant zu RemoveEmptyJoinInputLeft bezogen auf den rechten Eingabedatenfluss. □

Transformationsregel 4.12 – RemoveEmptyUnionInput1
Definitionsbereich: $o \in UO \subset OP$

Voraussetzung: Seien $(i_1, i_2) := o.\text{input}$; $o_1 := ProvOp(i_1)$ und $o_2 := ProvOp(i_2)$. Es gelte:

i) $o_1 \in GPO$
ii) $o_1.\text{triplePatterns} = ()$
iii) $o_1.\text{constraints} = ()$
iv) $ds \in GSO \rightarrow \mathsf{DS}(o_1) \neq \mathsf{DS}(o)$ wobei $ds \in \mathsf{DS}(o_1)$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_2.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_2.\text{optional}\big)$
iii) $o_2.\text{output} := \big(o_2.\text{output} \setminus \{i_2\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_1, i_2, o_1.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_1\}$

Erläuterung: Wenn für einen Vereinigungs-Operator o der bereitstellende Operator o_1 des ersten Eingabedatenflusses i_1 ein leerer Graphmuster-Operator ist, so wird die Vereinigung o aus o_1 und o_2 durch o_2 ersetzt. Demzufolge werden o und o_1 und deren Verbindungen untereinander aus dem SQGM entfernt und die Eigenschaften weiterer betroffender Datenflüsse angepasst. Die einzige Ausnahme bilden leere Graphmuster-Operatoren, die RDF-Graph-Klauseln mit einer Variablen als Graphspezifikation repräsentieren. Deren Entfernung verhindert die Voraussetzung iv).

Begründung: Der leere Graphmuster-Operator o_1 liefert keine Variablenbindungen und demzufolge auch keine weiteren Lösungen. Deshalb hat die Entfernung von o und o_1 keinen Einfluss auf das Ergebnis der Anfrage. □

Transformationsregel 4.13 – RemoveEmptyUnionInput2
Definitionsbereich: $o \in UO \subset OP$

Voraussetzung: Seien $(i_1, i_2) := o.\text{input}$; $o_1 := ProvOp(i_1)$ und $o_2 := ProvOp(i_2)$. Es gelte:

i) $o_2 \in GPO$
ii) $o_2.\text{triplePatterns} = ()$
iii) $o_2.\text{constraints} = ()$
iv) $ds \in GSO \rightarrow \mathsf{DS}(o_2) \neq \mathsf{DS}(o)$ wobei $ds \in \mathsf{DS}(o_2)$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_1.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_1.\text{optional}\big)$
iii) $o_1.\text{output} := \big(o_1.\text{output} \setminus \{i_1\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_1, i_2, o_2.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_2\}$

Erläuterung: Diese Regel ist das Pendant zu RemoveEmptyUnionInput1 bezogen auf den zweiten Eingabedatenfluss. □

Für die in diesem Abschnitt spezifizierten Regeln können keine Abhängigkeiten in Bezug auf mögliche Folgeregeln angegeben werden, weil nach der Entfernung eines leeren Graphmuster-Operators, dieser nicht weiter behandelt werden muss. Um alle möglichen Anwendungen für die Regeln zu finden, müssen alle Operatoren der oberen Ebene des SQGM mittels einer Tiefensuche rekursiv auf Anwendbarkeit untersucht werden. Eine vollständige Anwendung dieser Strategie führt zu einem SQGM, welches die vierte Eigenschaft der Normalform, die Freiheit von leeren Graphmuster-Operatoren, erfüllt.

4.2.5 Vergleich mit anderen Normalformen

Die spezifizierten Eigenschaften eines normalisierten SQGM spiegeln einerseits die Besonderheiten der SPARQL-Spezifikation und andererseits die Merkmale des SQGM wieder. Demzufolge sind sie nicht mit den Eigenschaften von Normalformen der relationalen Algebra vergleichbar.

Im Rahmen von SPARQL-Anfragen werden in [PAG06] zwei Normalformen angegeben. Übertragen auf das SQGM würde die erste fordern, dass sich in der Zusammenführungsschicht (siehe Abschnitt 4.1.5 auf Seite 50) alle Vereinigungs-Operatoren überhalb der Verbund-Operatoren befinden. Ein entsprechend geformtes SQGM erzwingt, dass Verbundoperationen immer vor den Vereinigungen ausgeführt werden. Im Rahmen dieser Arbeit wird eine solche Umwandlung als Heuristik gesehen und in Abschnitt 6.6 (ab Seite 113) als solche untersucht. Diese Ansicht resultiert aus der Vermutung, dass die vorgegebene Ausführungsreihenfolge nicht bei allen Speicher- und Zugriffsmodellen Vorteile bezüglich der in Abschnitt 3.3.2 (ab Seite 35) vorgestellten Ziele der logischen Optimierung bringt. In diesen Fällen wäre die vorgeschlagene Normalform ungeeignet, denn während der logischen Optimierung müssten unter Umständen die Umformungen zur Normalisierung rückgängig gemacht werden.

Die zweite in [PAG06] angegebene Normalform betrifft Anfragen ohne Graphmuster-Vereinigungen und sie fordert, bezogen auf das SQGM, zuerst Verbund-Operatoren ohne optionale Eingänge und danach alle Verbund-Operatoren mit einem optionalen Eingang auszuführen. Da auch diese Umwandlung eine bestimmte Ausführungsreihenfolge anstrebt, wird sie hier ebenfalls als Heuristik gesehen.

4.3 Vereinfachung

Bei der Bearbeitung von Datenbankanfragen werden die normalisierten Interndarstellungen vor der Optimierung vereinfacht. Das Ziel dieser Vereinfachung ist die Entfernung von Redundanzen und Inkonsistenzen. So können während der Ausführung unnötige Operationen vermieden werden. Eine solche Vereinfachung kann auch auf ein normalisiertes SQGM angewandt werden. In diesem Rahmen werden äquivalente Operatoren, Tripelmuster und Wertebeschränkungen als Redundanzen und widersprüchliche Graphmuster-Operatoren als Inkonsistenzen verstanden. Auf beide Punkte wird im Folgenden näher eingegangen.

4.3.1 Entfernung von Redundanzen

Ein SQGM kann auf zwei Ebenen Redundanzen enthalten. In Bezug auf ein gesamtes SQGM können mehrere Operatoren redundant sein und bezogen auf einzelne Operatoren könnten diese jeweils Redundanzen enthalten.

Zwei oder mehr Operatoren werden als redundant bezeichnet, wenn sie äquivalent sind. Dazu müssen sie vom gleichen Typ sein und die gleichen Eigenschaften haben. Weiterhin müssen ihre Eingangsdatenflüsse äquivalent sein. Um ein SQGM von redundanten Operatoren zu befreien, müssten diese zu jeweils einem Operator verschmolzen werden. Die Menge der Ausgabedatenflüsse dieser verschmolzenen Operatoren würde sich durch die Vereinigung der Ausgabedatenflüsse der einzelnen, äquivalenten Operatoren ergeben. Demzufolge würde das SQGM nach der Redundanz-Entfernung Operatoren mit mehreren Ausgängen enthalten. Das verletzt jedoch die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52), welche hauptsächlich während der Restrukturierung von Verbund- und Vereinigungsbäumen vorausgesetzt wird. Deshalb müssen redundante Operatoren während der Vereinfachung eines SQGM beibehalten werden. Es sei jedoch darauf hingewiesen, dass während der physischen Optimierung, nach der vollständigen Restrukturierung des SQGM, durchaus redundante Operatoren auf die beschriebene Weise entfernt werden könnten, was eventuell zu einer Reduzierung der auszuführenden Operationen beiträgt.

Redundanzen innerhalb von Operatoren können nur bei Graphmuster-Operatoren auftreten. Hier könnten Tripelmuster oder Wertebeschränkungen äquivalent sein, die Ausdrücke von Wertebeschränkungen könnten Redundanzen oder Idempotenzen beinhalten und einzelne Wertebeschrän-

kungen könnten sich lediglich in der Stärke ihrer Einschränkung unterscheiden. Die folgenden Regeln dienen zur Entfernung dieser Art von Redundanzen in einem SQGM.

Transformationsregel 4.14 – RemoveEqualTriplePatterns
Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Seien $t_1 \in o.\text{triplePatterns}$ und $t_2 \in o.\text{triplePatterns}$ zwei unterschiedliche Tripelmuster, wobei

i) t_1 und t_2 sind gleich

gilt. Zwei Tripelmuster sind gleich, wenn ihre Subjekte, Prädikate und Objekte jeweils gleich sind.

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{triplePatterns} := \big(t \mid t \in o.\text{triplePatterns} \wedge t \neq t_2\big)$

Erläuterung: Wenn ein Graphmuster-Operator o zwei gleiche Tripelmuster t_1 und t_2 enthält, so wird eines entfernt.

Begründung: Diese Regel behält die Semantik des SQGM bei, da für t_1 und t_2 auf Grund der Gleichheit dieselben Musterübereinstimmungen gefunden werden und deshalb das Entfernen von t_2 nicht die Menge der von o erzeugten Variablenbindungen beeinflusst. □

Transformationsregel 4.15 – RemoveEquivConstraints
Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Seien $c_1 \in o.\text{constraints}$ und $c_2 \in o.\text{constraints}$ zwei unterschiedliche Wertebeschränkungen und es gilt:

i) die Ausdrücke von c_1 und c_2 sind äquivalent

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{constraints} := \big(c \mid c \in o.\text{constraints} \wedge c \neq c_2\big)$

Erläuterung: Wenn ein Graphmuster-Operator o zwei Wertebeschränkungen c_1 und c_2 mit äquivalenten Ausdrücken enthält, so wird eine entfernt.

Begründung: Diese Regel behält die Semantik des SQGM bei, da c_1 und c_2 auf Grund der Äquivalenz die gleichen Beschränkungen der Lösungsmenge bewirken. □

Transformationsregel 4.16 – RemoveLessRestrConstraint
Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Seien $c_1 \in o.\text{constraints}$ und $c_2 \in o.\text{constraints}$ zwei unterschiedliche Wertebeschränkungen und es gilt:

i) c_1 ist stärker als c_2

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{constraints} := \big(c \mid c \in o.\text{constraints} \wedge c \neq c_2\big)$

Erläuterung: Wenn der Graphmuster-Operator o zwei Wertebeschränkungen c_1 und c_2 enthält, welche sich lediglich in der Stärke ihrer Einschränkung unterscheiden, so wird die schwächere Wertebeschränkungen entfernt. Beispielsweise ist die Beschränkung

$$c_1 = (?x > 5)$$

stärker als

$$c_2 = (?x > 2)$$

und deshalb kann c_2 entfernt werden. Es sei darauf hingewiesen, dass der Vergleich zweier Wertebeschränkung bezüglich der Stärke ihrer Einschränkung nicht trivial ist.

Begründung: Diese Regel behält die Semantik des SQGM bei, da alle durch c_2 ausgeschlossenen Lösungen auch durch c_1 ausgeschlossen werden. □

Transformationsregel 4.17 – SimplifyIdempConstraint

Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Sei $c \in o.\text{constraints}$ eine Wertebeschränkung und es gilt:

i) der Ausdruck von c kann vereinfacht werden

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ durch die Transformation

i) $o.\text{constraints} := \big(c^* \mid c^* \in o.\text{constraints} \wedge c^* \neq c\big) \circ \big(c'\big)$

wobei c' aus c durch die Vereinfachung des Ausdrucks entsteht.

Erläuterung: Wenn der Graphmuster-Operator o eine Wertebeschränkung c enthält, deren Ausdruck vereinfacht werden kann, so wird c durch c' mit dem vereinfachten Ausdruck ersetzt. Beispielsweise könnte ein boolescher Ausdruck der Form

$$a \vee a$$

durch

$$a$$

ersetzt werden. Durch eine Kenntnis des Schemas der abgefragten RDF-Datenmenge könnten sich zusätzliche Vereinfachungen ergeben. Es sei darauf hingewiesen, dass die Vereinfachung der Ausdrücke von Wertebeschränkung im Rahmen von SPARQL nicht trivial ist.

Begründung: Diese Regel behält die Semantik des SQGM bei, da c und c' auf Grund der Äquivalenz die gleichen Beschränkungen der Lösungsmenge bewirken. □

Transformationsregel 4.18 – RemoveIdempConstraint

Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Sei $c \in o.\text{constraints}$ eine Wertebeschränkung und es gilt:

i) der Ausdruck von c ist immer wahr

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{constraints} := \big(c^* \mid c^* \in o.\text{constraints} \wedge c^* \neq c\big)$

Erläuterung: Wenn der Graphmuster-Operator o eine Wertebeschränkung c enthält, deren Ausdruck immer wahr ist, so wird c aus o entfernt. Beispielsweise ist der boolesche Ausdruck

$$a \vee \neg a$$

immer wahr. Durch eine Kenntnis des Schemas der abgefragten RDF-Datenmenge könnten sich zusätzliche Möglichkeiten für immer wahre Ausdrücke ergeben. Es sei darauf hingewiesen, dass das Erkennen von solchen Ausdrücken im Rahmen von SPARQL nicht trivial ist.

Begründung: Diese Regel behält die Semantik des SQGM bei, da c keine Beschränkung der Lösungsmenge bewirkt. □

Die Tabelle 4.3 stellt die Abhängigkeiten der in diesem Abschnitt angegebenen Regeln in Bezug auf mögliche Folgeregeln dar. Deren Anwendbarkeit ist begrenzt, da ein SQGM nur eine endliche Anzahl von Graphmuster-Operatoren und jeder Graphmuster-Operator nur eine endlich große Menge von Tripelmustern und Wertebeschränkungen enthält. Eine vollständige Anwendung dieser Regeln führt zu einem vereinfachten SQGM dessen Graphmuster-Operatoren keine Redundanzen mehr enthalten.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verw. Op.
RemoveEqualTriplePatterns	o.triplePatterns $\neq$ ()	RemoveEqualTriplePatterns	o
RemoveEquivConstraints	o.constraints $\neq$ ()	RemoveEquivConstraints	o
RemoveLessRestrConstraint	o.constraints $\neq$ ()	RemoveLessRestrConstraint	o
SimplifyIdempConstraint	o.constraints $\neq$ ()	SimplifyIdempConstraint	o
	o.constraints $\neq$ ()	RemoveEquivConstraints	o
	o.constraints $\neq$ ()	RemoveLessRestrConstraint	o
	o.constraints $\neq$ ()	RemoveIdempConstraint	o
RemoveIdempConstraint	o.constraints $\neq$ ()	RemoveIdempConstraint	o

Tabelle 4.3: Abhängigkeiten zwischen den Transformationsregeln zum Entfernen von Redundanzen in Graphmuster-Operatoren

4.3.2 Behandlung von Widersprüchen

Ein SQGM kann Graphmuster-Operatoren mit widersprüchlichen Wertebeschränkungen beinhalten, deren Erkennung jedoch nicht trivial ist. Es ist möglich, dass die Ausdrücke einzelner Wertebeschränkungen für sich widersprüchlich sind. Andererseits könnten sich zwei oder mehr Wertebeschränkungen gegenseitig widersprechen. Einige Widersprüche könnten durch syntaktische Untersuchung der Ausdrücke erkannt werden. Andere Wertebeschränkungen könnten sich erst mit Bezug auf ein spezielles Schema als widersprüchlich herausstellen.

Die gefundenen Widersprüche können in dem SQGM in Richtung Wurzel propagiert werden, wobei ganze Teilbäume des SQGM widersprüchlich werden. Diese können dann aus dem SQGM entfernt werden. Das führt nicht nur zur Verkleinerung der Lösungsmenge sondern auch zu einer Verringerung der Anzahl auszuführender Operatoren und somit zu einer schnelleren Abarbeitung der möglichen Ausführungspläne. Deshalb ist es sinnvoll, widersprüchliche Wertebeschränkungen zu finden und zu behandeln.

Es sei an dieser Stelle darauf hingewiesen, dass ein nicht erkannter Widerspruch keinen Einfluss auf das Resultat einer Anfrage hat. Während ein erkannter und im SQGM nach oben propagierter Widerspruch dafür sorgt, dass Musterübereinstimmungen bzw. Variablenbindungen gar nicht erst gefunden werden, werden die ermittelten Variablenbindungen durch einen nicht erkannten Widerspruch wieder aus der Lösungsmenge entfernt.

Wegen des hohen Aufwandes der Bestimmung von Widersprüchen ist zwischen den notwendigen Ressourcen zum Finden und einer möglichen Einsparung von Ressourcen zur weiteren Bearbeitung und Ausführung der Anfrage abzuwägen. So könnte eine konkrete Implementierung bespielsweise lediglich eine kleine Gruppe von leicht zu erkennenden Mustern widersprüchlicher Ausdrücke suchen und weitere Widersprüche ignorieren.

Die folgenden Regeln dienen zum Finden und Propagieren von Widersprüchen. Wie bereits erwähnt, werden beim Propagieren einige Operatoren und Datenflüsse aus dem SQGM entfernt. Bei einer Implementierung ist zu beachten, dass die zu löschenden Datenflüsse aus den Eigenschaften *input* und *output* der, durch sie verbundenen Operatoren entfernt werden.

Transformationsregel 4.19 – FindContradiction

Definitionsbereich: $o \in GPO \subset OP$

Voraussetzung: Es gelte:

i) o.contr := F

ii) o.constraints $\neq$ ()

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) (o.constraints enthält einen Widerspruch) $\rightarrow$ o.contr := T

Erläuterung: Wenn ein Graphmuster-Operator o einen Widerspruch enthält, so wird dieser als widersprüchlich gekennzeichnet.

Begründung: Diese Regel behält die Semantik des SQGM bei, da sie lediglich einen widersprüchlichen Operator als solchen markiert. □

Transformationsregel 4.20 – HandleContrInput

Definitionsbereich: $o \in (SRO \cup DRO \cup CRO \cup ARO \cup SMO) \subset OP$

Voraussetzung: Es gelte:

i) $o.\text{input} \neq \text{undef.}$
ii) $\mathsf{ProvOp}(o.\text{input}) \in GPO$
iii) $\mathsf{ProvOp}(o.\text{input}).\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $DF' := DF \setminus \{o.\text{input}, \mathsf{ProvOp}(o.\text{input}).\text{input}\}$
ii) $OP' := OP \setminus \{\mathsf{ProvOp}(o.\text{input})\}$

Erläuterung: Wenn sich im SQGM direkt unterhalb des Operators o ein widersprüchlicher Graphmuster-Operator befindet, so wird dieser und dessen Verbindung zu o aus dem SQGM entfernt.

Begründung: Ein widersprüchlicher Graphmuster-Operator leistet keinen Beitrag zur Lösungsmenge. Wenn ein solcher Operator den Eingangsdatenfluss des Lösungsanpassungs- oder ergebnis-erzeugenden Operators o bereitstellt, dann ist die Lösungsmenge der Anfrage leer. Das trifft auch zu, wenn o keinen Eingangsdatenfluss hat. Demzufolge kann der widersprüchliche Operator aus dem SQGM entfernt werden, ohne dessen Semantik zu verändern. □

Transformationsregel 4.21 – HandleContrJoinInputLeft

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{left}}.\text{optional} = \mathsf{F}$
ii) $o_{\text{left}} \in GPO$
iii) $o_{\text{left}}.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ durch die Transformation

i) $o_{\text{left}}.\text{output} := o.\text{output}$
ii) $o.\text{output} := ()$
iii) $DF' := DF \setminus \left(\{i_{\text{left}}, i_{\text{right}}\} \cup SubDFs_{o_{\text{right}}}\right)$
iv) $OP' := OP \setminus \left(\{o, o_{\text{right}}\} \cup SubVPOPs_{o_{\text{right}}}\right)$

wobei $SubVPOPs_{o_{\text{right}}}$ die Menge aller unterhalb von o_{right} befindlichen variablen-bereitstellenden Operatoren ist und $SubDFs_{o_{\text{right}}}$ die Menge aller Eingangs-Datenflüsse der Operatoren aus $SubVPOPs_{o_{\text{right}}}$.

Erläuterung: Wenn für einen Verbund-Operator o der bereitstellende Operator o_{left} des linken, nicht optionalen Eingangsdatenflusses i_{left} ein widersprüchlicher Graphmuster-Operator ist, so wird dieser nach oben propagiert. Hierbei wird der Verbund o aus o_{left} und o_{right} durch den widersprüchlichen o_{left} ersetzt und o und der gesamte rechte Teilbaum werden aus dem SQGM entfernt.

Begründung: Die widersprüchlichen Wertebeschränkungen in o_{left} müssen auch für die, aus der Verbundoperation entstehenden Lösungen gelten. Demzufolge kann der gesamte Verbund als widersprüchlich angesehen werden. Die Semantik des SQGM bleibt bei der Ersetzung des widersprüchlichen Verbundes durch den widersprüchlichen Graphmuster-Operator o_{left} erhalten.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Transformationsregel 4.22 – HANDLECONTRJOININPUTRIGHT

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{right}}.\text{optional} = \mathsf{F}$
ii) $o_{\text{right}} \in GPO$
iii) $o_{\text{right}}.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ durch die Transformation

i) $o_{\text{right}}.\text{output} := o.\text{output}$
ii) $o.\text{output} := ()$
iii) $DF' := DF \setminus (\{i_{\text{left}}, i_{\text{left}}\} \cup SubDFs_{o_{\text{left}}})$
iv) $OP' := OP \setminus (\{o, o_{\text{left}}\} \cup SubVPOPs_{o_{\text{left}}})$

wobei $SubVPOPs_{o_{\text{left}}}$ die Menge aller unterhalb von o_{left} befindlichen variablen-bereitstellenden Operatoren ist und $SubDFs_{o_{\text{left}}}$ die Menge aller Eingangs-Datenflüsse der Operatoren aus $SubVPOPs_{o_{\text{left}}}$.

Erläuterung: Diese Regel ist das Pendant zu HANDLECONTRJOININPUTLEFT bezogen auf den rechten Eingangsdatenfluss.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Transformationsregel 4.23 – HANDLECONTRJOINOPTINPUTLEFT

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{left}}.\text{optional} = \mathsf{T}$
ii) $o_{\text{left}} \in GPO$
iii) $o_{\text{left}}.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_{\text{right}}.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_{\text{right}}.\text{optional}\big)$
iii) $o_{\text{right}}.\text{output} := \big(o_{\text{right}}.\text{output} \setminus \{i_{\text{right}}\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_{\text{left}}, i_{\text{right}}, o_{\text{left}}.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_{\text{left}}\}$

Erläuterung: Wenn für einen Verbund-Operator o der bereitstellende Operator o_{left} des linken, optionalen Eingangsdatenflusses i_{left} ein widersprüchlicher Graphmuster-Operator ist, so wird der bereitstellende Operator o_{right} des rechten Eingabedatenflusses i_{right} nach oben gezogen. Hierbei wird der Verbund o aus o_{left} und o_{right} durch o_{right} ersetzt und o und der widersprüchliche o_{left} aus dem SQGM entfernt.

Begründung: Die widersprüchlichen Wertebeschränkungen in o_{left} müssen nicht für die, aus der Verbundoperation entstehenden Lösungen gelten, weil o_{left} in diese nur als optionaler Teil eingeht. Da o_{left} wegen der Widersprüchlichkeit keinen Beitrag zur Lösungsmenge liefert, sind die Lösungen der Verbundoperation genau die Lösungen des rechten Operators o_{right}. Demzufolge kann der gesamte Verbund o durch o_{right} ersetzt werden ohne die Semantik des SQGM zu verändern. □

Transformationsregel 4.24 – HandleContrJoinOptInputRight

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{right}}.\text{optional} = \mathsf{T}$
ii) $o_{\text{right}} \in GPO$
iii) $o_{\text{right}}.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_{\text{left}}.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_{\text{left}}.\text{optional}\big)$
iii) $o_{\text{left}}.\text{output} := \big(o_{\text{left}}.\text{output} \setminus \{i_{\text{left}}\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_{\text{left}}, i_{\text{right}}, o_{\text{right}}.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_{\text{right}}\}$

Erläuterung: Diese Regel ist das Pendant zu HandleContrJoinOptInputLeft bezogen auf den rechten Eingangsdatenfluss. □

Transformationsregel 4.25 – HandleContrUnionInput1

Definitionsbereich: $o \in UO \subset OP$

Voraussetzung: Seien $(i_1, i_2) := o.\text{input}$; $o_1 := ProvOp(i_1)$ und $o_2 := ProvOp(i_2)$. Es gelte:

i) $o_1 \in GPO$
ii) $o_1.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_2.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_2.\text{optional}\big)$
iii) $o_2.\text{output} := \big(o_2.\text{output} \setminus \{i_2\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_1, i_2, o_1.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_1\}$

Erläuterung: Wenn für einen Vereinigungs-Operator o der bereitstellende Operator o_1 des ersten Eingangsdatenflusses i_1 ein widersprüchlicher Graphmuster-Operator ist, so wird die Vereinigung o aus o_1 und o_2 durch o_2 ersetzt. Demzufolge werden o und o_1 und deren Verbindungen untereinander aus dem SQGM entfernt und die Eigenschaften weiterer betroffender Datenflüsse angepasst.

Begründung: Der Widerspruch in o_1 hat keinen Einfluss auf die Lösungen aus o_2. Weil o_1 keine Variablenbindungen liefert, sind die Lösungen der Vereinigung genau die Variablenbindungen von o_2. Demzufolge kann die Vereinigung o durch o_2 ersetzt werden, ohne dass die Semantik des SQGM verändert wird. □

Transformationsregel 4.26 – HandleContrUnionInput2

Definitionsbereich: $o \in UO \subset OP$

Voraussetzung: Seien $(i_1, i_2) := o.\text{input}$; $o_1 := ProvOp(i_1)$ und $o_2 := ProvOp(i_2)$. Es gelte:

i) $o_1 \in GPO$
ii) $o_1.\text{contr} = \mathsf{T}$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $\forall\, d \in o.\text{output} : d.\text{vars} := o_1.\text{provVars}$
ii) $\forall\, d \in o.\text{output} : d.\text{optional} := \big(d.\text{optional} \vee i_1.\text{optional}\big)$
iii) $o_1.\text{output} := \big(o_1.\text{output} \setminus \{i_1\}\big) \circ o.\text{output}$
iv) $o.\text{output} := ()$
v) $DF' := DF \setminus \{i_1, i_2, o_2.\text{input}\}$
vi) $OP' := OP \setminus \{o, o_2\}$

Erläuterung: Diese Regel ist das Pendant zu HANDLECONTRUNIONINPUT1 bezogen auf den zweiten Eingangsdatenfluss. □

Die Tabelle 4.4 stellt die Abhängigkeiten der in diesem Abschnitt angegebenen Regeln in Bezug auf mögliche Folgeregeln dar. Eine vollständige Anwendung dieser Regeln führt zu einem vereinfachten SQGM. Dieses enthält keine als widersprüchlich gekennzeichneten Graphmuster-Operatoren und keine der mittels FINDCONTRADICTION bestimmbaren Widersprüche. Ein entsprechender Algorithmus ist deterministisch, weil einerseits der Regel FINDCONTRADICTION für die Überprüfung nur eine endlich große Menge von Graphmuster-Operatoren mit jeweils einer endlich großen Menge von Wertebeschränkungen zur Verfügung steht und die gefundenen Widersprüche andererseits in Richtung der Wurzel des SQGM propagiert werden.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verw. Oper.
FindContradiction	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in (SRO \cup DRO \cup CRO \cup ARO \cup SMO)$	HandleContrInput	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput1	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput2	$\mathsf{UsingOp}(d)$
HandleContrJoinInputLeft	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in (SRO \cup DRO \cup CRO \cup ARO \cup SMO)$	HandleContrInput	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{left}}.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput1	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput2	$\mathsf{UsingOp}(d)$
HandleContrJoinInputRight	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in (SRO \cup DRO \cup CRO \cup ARO \cup SMO)$	HandleContrInput	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputLeft	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in JO$	HandleContrJoinOptInputRight	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput1	$\mathsf{UsingOp}(d)$
	$\forall\, d \in o_{\text{right}}.\text{output} : \mathsf{UsingOp}(d) \in UO$	HandleContrUnionInput2	$\mathsf{UsingOp}(d)$

Tabelle 4.4: Abhängigkeiten zwischen den Transformationsregeln zum Finden und Propagieren widersprüchlicher Graphmuster-Operatoren

Kapitel 5

Implementierung der Konzepte

Die in den vorherigen beiden Kapiteln vorgestellten Datenstrukturen und theoretischen Konzepte wurden im Rahmen der Diplomarbeit auch praktisch realisiert und auf Anwendbarkeit überprüft. Eine entsprechende Implementierung ermöglicht zusätzlich eine praktische Untersuchung möglicher Heuristiken.

In diesem Kapitel wird die prototypisch entwickelte Implementierung vorgestellt. Sie basiert auf dem in Java entwickelten Semantic-Web-Framework Jena und dessen Anfragesystem ARQ, welche im ersten Abschnitt vorgestellt werden. Der zweite Abschnitt beschreibt das entworfene Klassenmodell zur Repräsentation des im Kapitel 3 spezifizierten SQGM und seiner Bestandteile.

Um diese Datenklassen für eine prototypische Realisierung der diskutierten Anfrageumwandlungen nutzen zu können, muss das implementierte SQGM in die Anfrageausführung von ARQ eingebunden werden. Die Möglichkeiten für eine solche Einbettung und die tatsächlich realisierte Erweiterung der Anfrageausführung werden im dritten Abschnitt vorgestellt.

Der vierte Abschnitt dieses Kapitels beschreibt die Implementierung eines Systems zur Regelausführung. Dieses System bildet die Grundlage für die regelbasierte Restrukturierung. Die hierfür entwickelte Restrukturierungskomponente wird im fünften Abschnitt vorgestellt.

Der sechste Abschnitt schließt dieses Kapitels mit einer Beschreibung der implementierten Testumgebung ab. Diese kann zur Ausführung von Regressionstests und zur Erfassung der Bearbeitungszeit einzelner Anfragen genutzt werden.

5.1 Die Basis – Jena und ARQ

Um die prototypisch entwickelte Implementierung in die Anfragebearbeitung von Jena und dessen Anfragesystem ARQ einordnen zu können, stellt dieser Abschnitt Jena und ARQ kurz vor. Eine ausführlichere Beschreibung der Architektur von Jena findet sich in [CDD+03] und der Jena-Dokumentation[1]. ARQ wird in [Sea06] stichpunktartig vorgestellt und die ARQ-Dokumentation[2] enthält Informationen zur Verwendung des Anfragesystems.

Die Architektur von Jena setzt sich aus drei aufeinander aufbauenden Schichten zusammen (siehe Abbildung 5.1). Das sind die *Graph-Schicht*, die *erweiterte Graph-Schicht* und eine *Schicht zur Präsentation*, welche alternativ in Form einer Modell- oder einer Ontologie-Schicht auftritt.

Die Grundlage bildet die Graph-Schicht. Sie basiert auf der in [KC04] spezifizierten RDF-Syntax und definiert Schnittstellen zur Repräsentation von RDF-Graphen. Die wichtigsten Objekte innerhalb dieser Schicht sind Graphen, Tripel und Knoten, welche durch die Schnittstellenbeschreibungen `Graph`, `Triple` und `Node` repräsentiert sind. Jena stellt mehrere Implementierungen der Graph-Schicht zur Verfügung. Einerseits kann auf konkrete, materialisierte RDF-Graphen in Form von Tripelmengen im Hauptspeicher oder in Datenbanken zugegriffen werden. Andererseits sind

[1] `http://jena.sourceforge.net/documentation.html`
[2] `http://jena.sourceforge.net/ARQ/documentation.html`

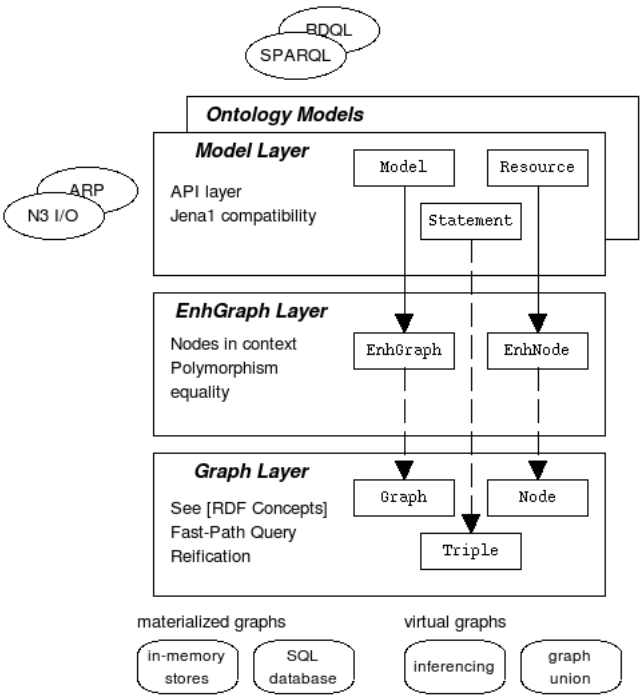


Abbildung 5.1: Die Jena-Architektur nach [CDD+03]

virtuelle RDF-Graphen möglich, deren Tripel durch Inferenz aus einer Basisdatenmenge gefolgert werden oder aus der Vereinigung zweier RDF-Graphen entstehen.

Die auf der Graph-Schicht aufbauende erweiterte Graph-Schicht bildet eine Zwischenebene, welche verschiedene Sichten auf die Graphen und Knoten der Graph-Schicht anbietet. In den oberen Schichten können mehrere dieser unterschiedlichen Betrachtungsweisen gleichzeitig verwendet werden. Dadurch wird in Jena Mehrfachvererbung und Typisierung der Knoten ermöglicht.

Die Präsentationsschichten stellen Abstraktionen der RDF-Graphen dar, welche bei der Entwicklung von Anwendungen genutzt werden können. Mit der Modell-API und der Ontologie-API stellt Jena zwei konkrete Ausprägungen der Präsentationsschicht zur Verfügung. Die Modell-Schnittstelle basiert auf den Abstraktionen in [BG04]. RDF-Datenmengen werden hier als Modelle bezeichnet und beinhalten Aussagen über Ressourcen. Die entsprechenden Objekte werden durch die Schnittstellenbeschreibungen `Model`, `Statement` und `Resource` und deren Spezialisierungen repräsentiert. Alternativ bietet die Ontologie-Schnittstelle eine Unterstützung der Konzepte von Ontologien. Die wichtigsten Objekte hierbei sind Klassen, Eigenschaften und Individuen, repräsentiert durch die Schnittstellenbeschreibungen `OntClass`, `OntProperty` und `Individual` und deren Spezialisierungen.

Für die Ausführung von Anfragen an die verwalteten RDF-Datenmengen bietet Jena eine Unterstützung bei der Bestimmung von Musterübereinstimmungen einfacher Graphmuster an. Hierfür enthält die Schnittstellenbeschreibung der Graph-Schicht zwei Möglichkeiten. Die erste ist die Operation `find` der `Graph`-Schnittstelle. Sie implementiert die einfache Musterübereinstimmung eines einzelnen Tripelmusters und liefert einen Iterator über die Menge der, im repräsentierten RDF-Graph übereinstimmenden Tripel. Für die vollständige Ausführung einer SPARQL-Anfrage müsste diese in einzelne `find`-Anfragen zerlegt und die entsprechenden Lösungsmengen miteinander verbunden werden.

Die zweite Möglichkeit zur Auswertung von Anfragen ist die *Fast-Path-Anfrageausführung*, für die jede Implementierung der `Graph`-Schnittstelle ein `QueryHandler`-Objekt bereitstellt. Über dieses können einfache Graphmuster, also Mengen von Tripelmustern ausgewertet werden. Dieser

Ansatz hat den Vorteil, dass bei der gleichzeitigen Auswertung mehrerer Tripelmuster spezielle Eigenschaften der konkreten `Graph`-Implementierung ausgenutzt werden können. So unterteilt der Fast-Path-Algorithmus[3] für RDF-Daten in Datenbanken die Tripelmuster in einzelne Gruppen, welche zusammen durch jeweils eine einzige Datenbankanfrage ausgewertet werden können. Jede dieser Datenbankanfragen enthält Verbundoperationen zur Verknüpfung der Auswertung der einzelnen Tripelmuster der jeweiligen Gruppe. Damit wird ein Teil der Anfrageoptimierung dem DBMS übertragen. Für die meisten anderen Fälle bietet Jena standardmässig eine einfache `QueryHandler`-Implementierung an, die das einfache Graphmuster durch verschachtelte Aufrufe der `find`-Operation auswertet.

Wie bereits erwähnt ist ARQ das Anfragesystem zu Jena. Als vollständiges Framework zur Anfrageausführung beinhaltet es

- Datenstrukturen zur Repräsentation von Anfragen und Ergebnissen,
- einen SPARQL-Parser,
- Schnittstellenbeschreibungen für eine Anfragebearbeitungskomponente,
- eine Implementierung dieser Schnittstellen,
- Möglichkeiten zum Umgang mit den Ergebnissen einer Anfrage und
- ein Framework für Regressionstests.

Die Datenstrukturen bilden die Basis des gesamten Anfragesystems. Die Repräsentation von Anfragen ist, im Gegensatz zum SQGM, sehr stark an die, in [PS06] beschriebene Grammatik für SPARQL-Anfragen angelehnt. Der Parser erzeugt entsprechende Instanzen der Repräsentationsdatentypen.

Die implementierte Anfragebearbeitungskomponente baut auf der oben beschriebenen Musterübereinstimmung von einfachen Graphmustern in Jena auf und kann deshalb für jeden Jena-Graph und jedes Jena-Modell eingesetzt werden. Durch ihren modularen Aufbau kann sie erweitert werden. Es ist jedoch auch möglich, eine neue Anfragebearbeitungskomponente auf Basis der Schnittstellenbeschreibung zu implementieren und in ARQ zur Benutzung zu registrieren.

5.2 Implementierung des SQGM

Als Basis der prototypischen Realisierung wurden die in Kapitel 3 spezifizierten Typen von Operatoren und Datenflüssen mit ihren Eigenschaften modelliert und als Java-Klassen implementiert. Das Klassendiagramm in Abbildung 5.2 zeigt die entworfenen Klassen und deren Beziehungen.

Die abstrakte Basisklasse `Operator` repräsentiert alle SQGM-Operatoren. Spezialisierungen davon sind die abstrakten Klassen `GraphProvidingOperator` und `VariableProvidingOperator`, welche alle graph-bereitstellenden und variablen-bereitstellenden Operatoren repräsentieren. Von diesen sind verschiedene Klassen für die einzelnen Operatortypen abgeleitet.

Die beiden Arten von Datenflüssen wurden durch die Klassen `GraphFlow` und `VariableFlow` als Spezialisierungen der abstrakten Basisklasse `DataFlow` modelliert. Dabei werden die Eigenschaften *vars* und *optional* der Variablendatenflüsse in `VariableFlow` durch die Attribute `variables` und `optional` repräsentiert.

[3] `http://jena.sourceforge.net/DB/fastpath.html`

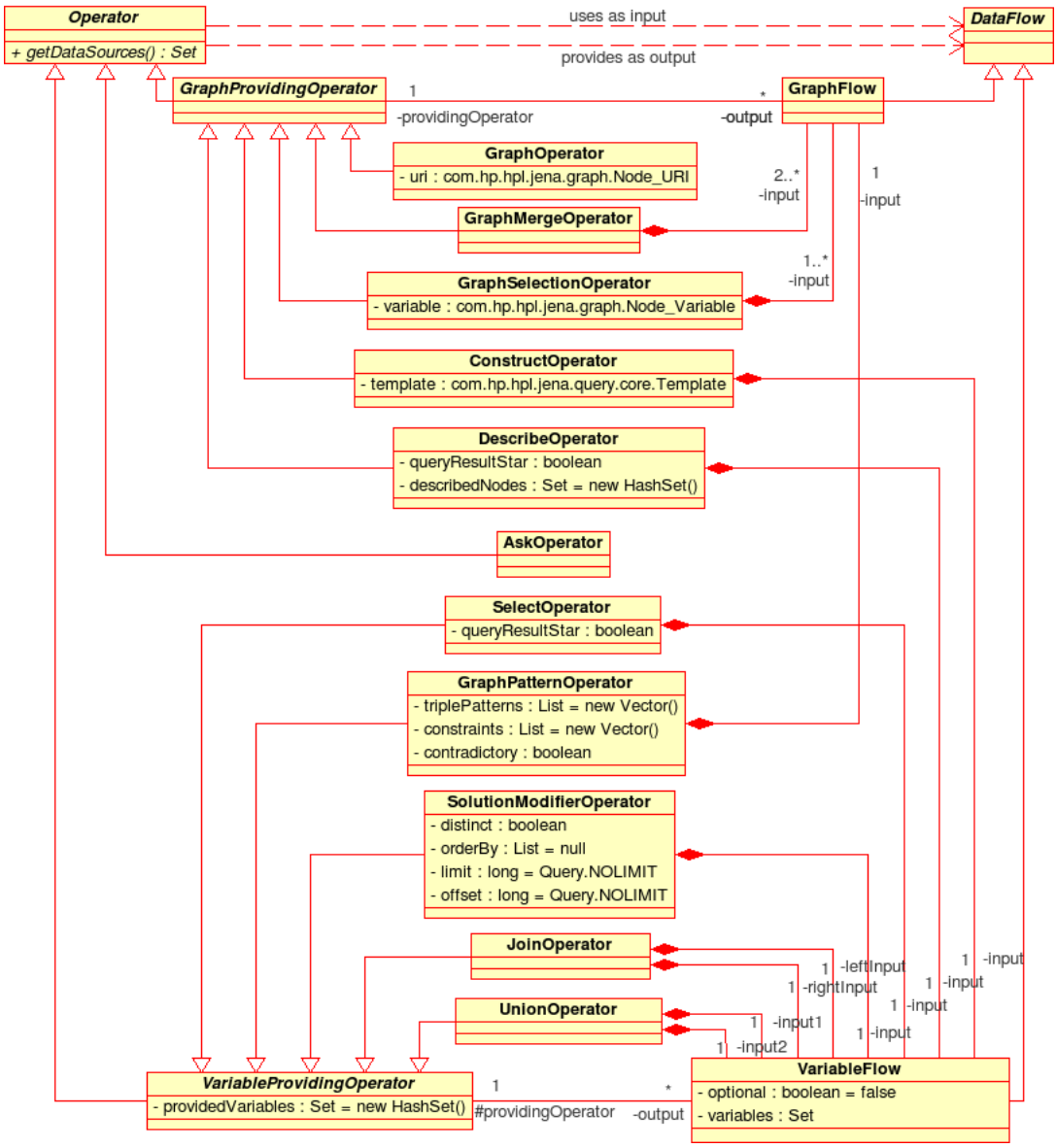


Abbildung 5.2: Entwurfsmodell der Klassen für Operatoren und Datenflüsse eines SQGM

Die Assoziation zwischen den Klassen `GraphProvidingOperator` und `GraphFlow` und zwischen `VariableProvidingOperator` und `VariableFlow` modellieren jeweils sowohl die Eigenschaft *output* der Operatoren als auch die Abbildung ProvOp. Anstatt auf der Ebene der Basisklassen `Operator` und `DataFlow` eine gemeinsame Assoziation zu modellieren, wurden zwei getrennte Assoziationen entworfen. Dadurch werden die Einschränkungen der Eigenschaft *output* und des Wertebereiches der Abbildung ProvOp abgebildet. Diese, in Abschnitt 3.1.2 auf Seite 23 definierten Einschränkungen beziehen sich auf die beiden verschiedenen Arten von Datenflüssen. Graphbereitstellende Operatoren dürfen lediglich Graphdatenflüsse bereitstellen und Variablendatenflüsse dürfen nur von variablen-bereitstellenden Operatoren ausgehen. Analog wurde die Eigenschaft *input* jeweils als Kompositionsattribut der einzelnen Operatortypen-Klassen modelliert.

Die speziellen Eigenschaften der einzelnen Operatortypen wurden als Attribute der zugehörigen Klassen modelliert. Beispielsweise entspricht das Attribut `uri` der Klasse `GraphOperator` der Eigenschaft *uri* von Graphoperatoren. Als Datentyp dieses Attributs wurde die Jena-Repräsentation eines URI-Knotens in RDF-Graphen gewählt.

Anfragen, welche durch ein SQGM repräsentiert werden, wurden durch die Klasse `Query` modelliert (siehe Abbildung 5.3). Hierbei wurde die Menge aller SQGM-Operatoren der Anfrage in mehrere Listen unterteilt und durch entsprechende Attribute modelliert. Dadurch ist es einfacher, auf Operatoren eines bestimmten Typs zuzugreifen. Die Operation `addOperator` fügt einen übergebenen Operator zur SQGM-Repräsentation der Anfrage hinzu und ordnet diesen Operator entsprechend seines Typs in eine der Listen ein. Der graph-bereitstellende Operator, welcher den Default-Graph der angefragten RDF-Datenmenge repräsentiert, wurde durch das Attribut `defaultGraph` modelliert.

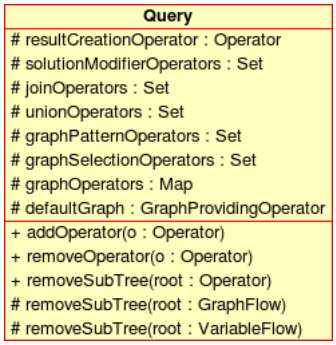


Abbildung 5.3: Entwurfsmodell einer SQGM-Repräsentation von SPARQL-Anfragen

5.3 Einbettung des SQGM in Jenas Anfragebearbeitung

Auf Basis des SQGM könnte eine alternative Anfragebearbeitungskomponente für ARQ entwickelt werden, welche unter anderem die in dieser Arbeit untersuchten Konzepte realisiert. Die Abbildung 5.4 stellt unter anderem die dafür zu implementierenden Schnittstellen dar. Über die ARQ-Klasse `QueryEngineRegistry` werden Objekte zur Erzeugung von Anfragebearbeitungskomponenten verwaltet. Während die Objekte zur Erzeugung die Schnittstelle `QueryEngineFactory` realisieren, müssen die erzeugten Komponenten der Schnittstellenbeschreibung `QueryExecution` genügen. Dementsprechend enthält ARQ für die bereitgestellte Anfragebearbeitungskomponente die Klassen `DefaultEngineFactory` und `QueryEngine` als Realisierungen der Schnittstellen. Analog dazu müssten für eine alternative Anfragebearbeitungskomponente auf Basis des SQGM die Klassen `QGMEngine` und `QGMEngineFactory` implementiert werden.

Während die in ARQ realisierte Komponente SPARQL-Anfragen bearbeitet, welche durch die ARQ-Klasse `Query` repräsentiert werden, verwendet die zu entwickelnde Klasse `QGMEngine` die SQGM-Repräsentation `QGMQuery`. Die Klasse `QGMQuery` ist eine Erweiterung der, im vorherigen

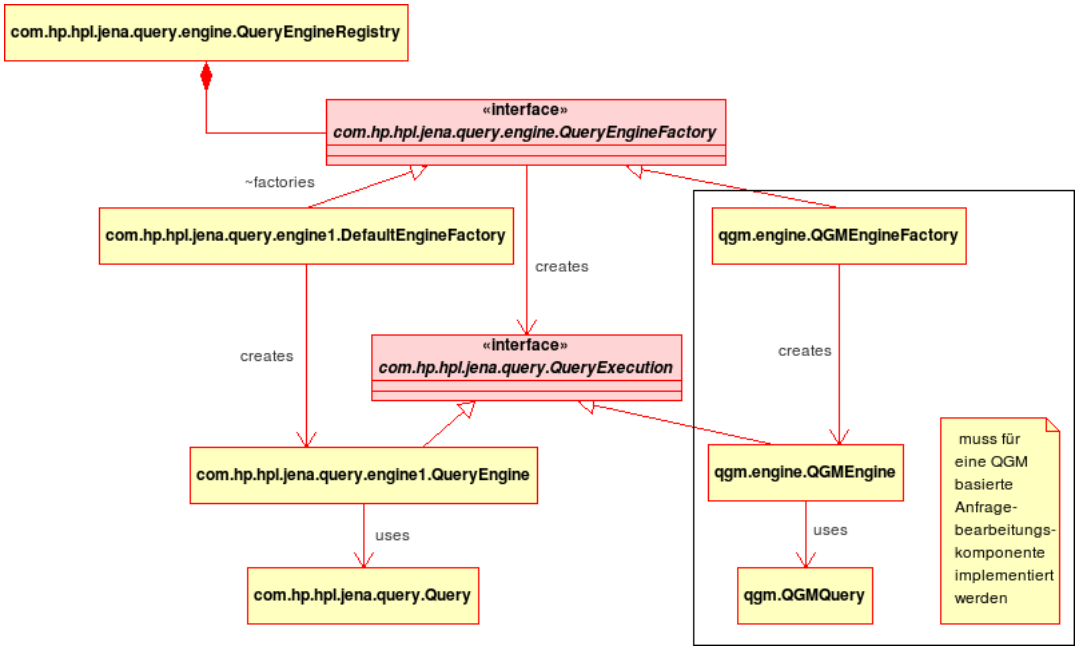


Abbildung 5.4: Entwurfsmodell für die Einbettung einer alternativen Anfragebearbeitungskomponente für ARQ auf Basis des SQGM

Abschnitt vorgestellen Klasse `Query` (siehe Abbildung 5.3). Sie enthält einige zusätzliche Attribute, um den Einsatz innerhalb des ARQ-Frameworks zu ermöglichen und die ARQ-Klasse `Query` im Rahmen der Anfragebearbeitung vollständig ersetzen zu können. Das Klassendiagramm in Abbildung 5.5 zeigt die Klasse `QGMQuery` und deren Beziehungen zu den bereits vorgestellten Klassen.

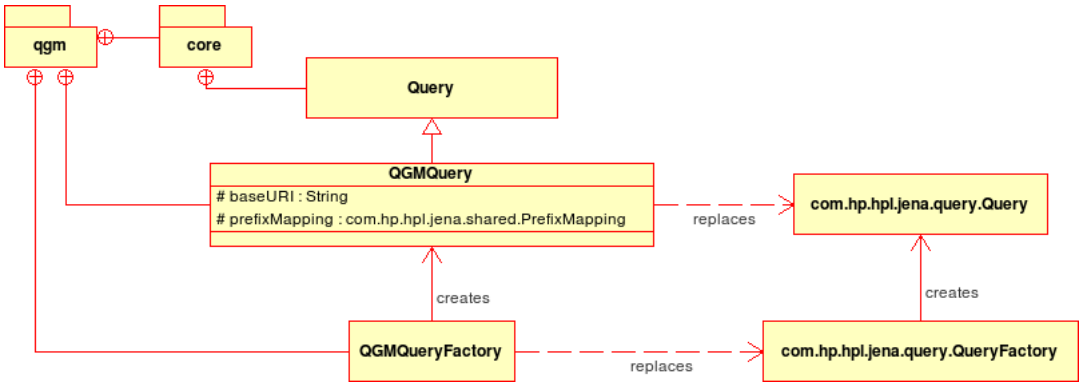


Abbildung 5.5: Entwurfsmodell zur Verdeutlichung der Beziehung zwischen der ARQ-Repräsentation von Anfragen und der Repräsentation auf Basis des SQGM

Zur Erzeugung des SQGM zu einer SPARQL-Anfrage wird der SPARQL-Parser von ARQ verwendet. Da dieser ein Objekt der ARQ-Klasse `Query` generiert, ist es lediglich notwendig, dieses Objekt in ein `QGMQuery`-Objekt zu transformieren. Da die Datentypen von ARQ zur Repräsentation einer Anfrage und ihrer Teile stark an die Grammatik für SPARQL-Anfragen angelehnt sind, konnte für diese Umwandlung das, in Abschnitt 4.1 vorgestellte Vorgehen zur Übersetzung einer SPARQL-Anfrage in ein SQGM mit leichten Anpassungen verwendet werden.

Der entsprechende Algorithmus wird durch die Klasse `QueryConverter` implementiert, welche in Abbildung 5.6 dargestellt ist. Das Klassendiagramm zeigt zusätzlich ihre Einordnung in

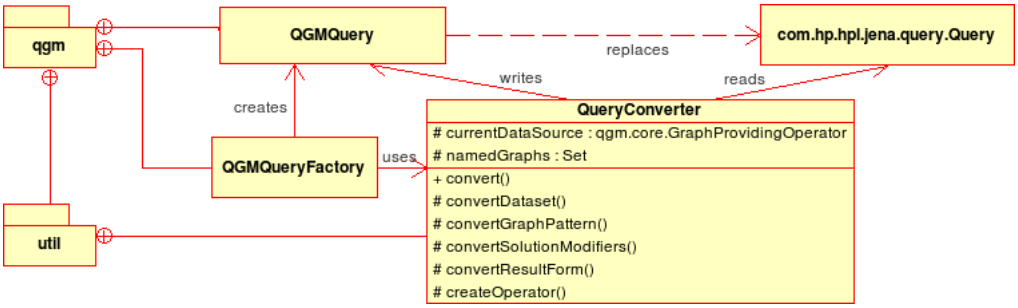


Abbildung 5.6: Entwurfsmodell zur Erzeugung von SQGM-Repräsentationen für Anfragen aus der ARQ-Repräsentationsform

die prototypische Implementierung. Entsprechend dem erwähnten Vorgehen werden hintereinander die Funktionen `convertDataset`, `convertGraphPattern`, `convertSolutionModifiers` und `convertResultForm` von der Hauptfunktion `convert` aufgerufen und dadurch das SQGM erzeugt. Die verschiedenen `createOperator`-Funktionen werden während der Ausführung der Funktion `convertGraphPattern` zur Modellierung des Graphmusters genutzt und implementieren den rekursiven Algorithmus CreateSQGMOperator (siehe Abbildung 4.1 auf Seite 42).

Um die in dieser Arbeit vorgestellten Regeln zur Normalisierung und Vereinfachung eines SQGM testen zu können und die Möglichkeit für die Untersuchung von Heuristiken zur Anfragerestrukturierung zu schaffen, ist keine vollständig implementierte, alternative Anfragebearbeitungskomponente notwendig. Demzufolge wurde lediglich die von ARQ bereitgestellte Klasse `QueryEngine` erweitert. Das Klassendiagramm in Abbildung 5.7 zeigt die prototypische Erweiterung.

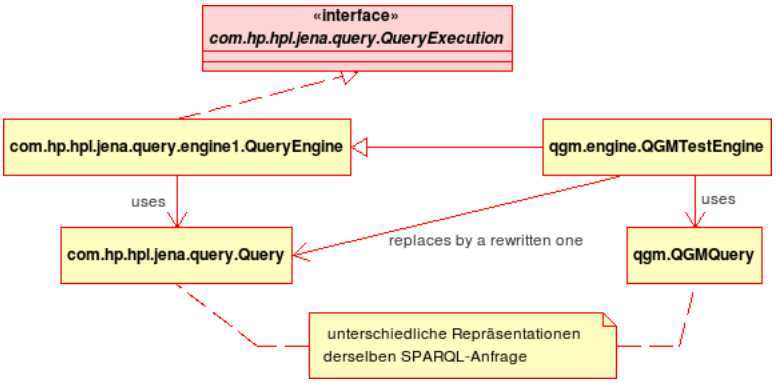


Abbildung 5.7: Entwurfsmodell der prototypischen Anfragebearbeitungskomponente als Erweiterung der ARQ-Realisierung

Im Gegensatz zum oben vorgestellten Entwurfsmodell für eine alternative Anfragebearbeitungskomponente (Abbildung 5.4 auf Seite 76) ist der entwickelte Prototyp eine Spezialisierung der ARQ-Realisierung und kann so deren Funktionalität zur Planerzeugung und Anfrageausführung nutzen. Er erweitert die ARQ-Komponente um die Möglichkeit, die in dieser Arbeit entwickelten Konzepte auf zu bearbeitende Anfragen anzuwenden. Dazu wird vor der Planerzeugung aus der ARQ-Repräsentation einer Anfrage mittels der oben beschrieben Klasse `QueryConverter` ein SQGM erzeugt. Dieses wird normalisiert, vereinfacht, restrukturiert und letztendlich wieder in eine ARQ-Repräsentation überführt. Die Abbildung 5.8 zeigt die Umwandlungen zwischen den Repräsentationsformen der Anfrage in diesem Zusammenhang.

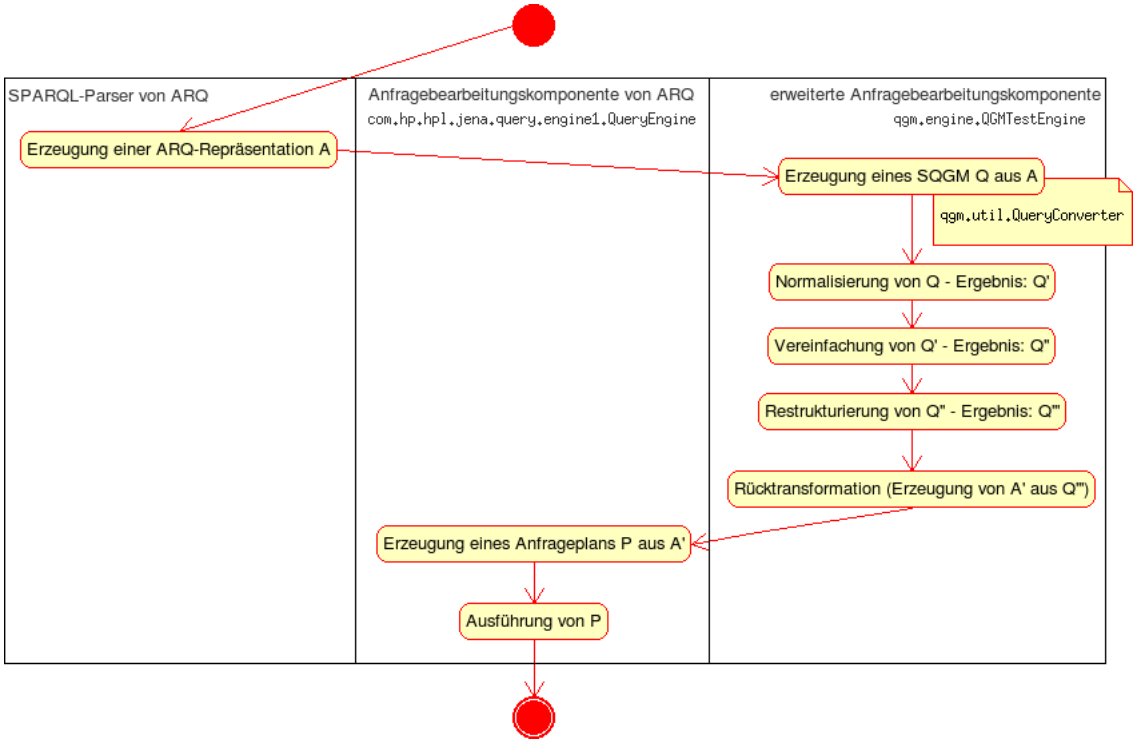


Abbildung 5.8: Anfragebearbeitung mit erweiterter Anfragebearbeitungskomponente

Nach der Anwendung der untersuchten Transformationsregeln kann die ARQ-Realisierung der Anfragebearbeitungskomponente mit einer ARQ-Repräsentation der transformierten Anfrage die Bearbeitung fortsetzen, wie das Aktivitätsdiagramm in Abbildung 5.8 zeigt. Auf diese Art ist es möglich, die in dieser Arbeit entwickelten Konzepte an SPARQL-Anfragen auf realen Datenmengen zu testen ohne eine vollständige Anfragebearbeitungskomponente implementieren zu müssen.

Um zu garantieren, dass die entwickelten Datenklassen des SQGM fehlerfrei sind und die implementierten Konzepte zur Anfrageumwandlung keinen Einfluss auf die ermittelte Ergebnismenge haben, wurde das Framework für Regressionstests von ARQ benutzt. In den Tests werden unterschiedlichste SPARQL-Anfragen ausgeführt und deren Ergebnisse jeweils mit den zu erwartenden Ergebnissen verglichen. Zur Ausführung der Tests unter Verwendung der erweiterten Anfragebearbeitungskomponente war es lediglich notwendig, diese, wie oben beschrieben, in dem System zur Verwaltung entsprechender Komponenten zu registrieren. Die Regressionstests werden fehlerfrei abgearbeitet.

5.4 Implementierung des Systems zur Regelausführung

Im Kapitel 4 wurden unter anderem Merkmale eines normalisierten und vereinfachten SQGM spezifiziert und Transformationsregeln zur Normalisierung und Vereinfachung angegeben. Diese Regeln wurden auf Basis der entwickelten Datenklassen (Abschnitt 5.2 ab Seite 73) zur Repräsentation des SQGM implementiert. Um die Regeln ergebnisorientiert abarbeiten zu können, musste zunächst jedoch ein System zur Regelausführung entworfen und implementiert werden, welches insbesondere die Abhängigkeiten der Regeln in Bezug auf mögliche Folgeregeln beachtet.

In diesem Abschnitt werden zuerst die Anforderung an das entwickelte Regelausführungssystem spezifiziert, danach wird das entworfene System beschrieben und abschließend werden die implementierte Normalisierung und die Vereinfachung als konkrete Anwendungen des Regelausführungssystems vorgestellt, wobei die Implementierung einer Beispielregel genauer diskutiert wird.

Zunächst werden jedoch die Begriffe *Transformationsregel* und *Regelinstanz* definiert, um deren Bedeutung im Rahmen dieser Arbeit festzulegen.

Definition 5.1 Eine *Transformationsregel* wandelt ein SQGM q in ein semantisch äquivalentes SQGM q' um. Jede Transformationsregel hat

- einen Namen,
- einen Wertebereich, welcher eine Teilmenge der Operatoren des zu transformierenden SQGM ist,
- eine optionale Menge an Voraussetzungen und
- eine nichtleere Liste mit Anweisungen zur Ausführung der Transformation.

Sowohl die Voraussetzungen als auch die Ausführungsanweisungen sind in Bezug auf den durch die Wertebereichsspezifikation festgelegten Operator formuliert und betreffen nur diesen und die direkt mit ihm verbundenen Operatoren. □

Definition 5.2 Eine *Regelinstanz* ist eine konkrete Transformationsregel, die sich auf einen individuellen Operator mit einem Typ aus dem Wertebereich der Regel bezieht. Dieser Operator wird als *betroffener Operator* (engl.: target operator) der Regelinstanz bezeichnet. Die Auführung einer Regelinstanz entspricht der Anwendung der Anweisungen zur Transformationsausführung auf den betroffenen Operator in der durch die Regeldefinition vorgegebenen Reihenfolge. Eine Regelinstanz kann nur ausgeführt werden, wenn die in der Regeldefinition angegebenen Voraussetzungen bezüglich des betroffenen Operators erfüllt sind. □

Definition 5.3 Eine Regelinstanz wird als *möglicherweise ausführbar* bezeichnet, wenn die in der Regeldefinition angegebenen Voraussetzungen noch nicht überprüft wurden. □

Definition 5.4 Eine Regelinstanz wird als *sicher ausführbar* bezeichnet, wenn die in der Regeldefinition angegebenen Voraussetzungen bezüglich des betroffenen Operators erfüllt sind. □

Beispiel 5.1 *Die Definition 4.14 auf Seite 63 beschreibt eine Transformationsregel mit dem Namen* RemoveEqualTriplePatterns. *Eine entsprechende Regelinstanz ist beispielsweise die* RemoveEqualTriplePatterns*-Regel, welche sich auf den Graphmuster-Operator des linken SQGM in Abbildung 5.9 bezieht. Dieser konkrete Operator enthält zwei gleiche Tripelmuster. Deshalb sind für ihn die Voraussetzungen in Definition 4.14 erfüllt und die Regelinstanz ist ausführbar. Die Ausführung überführt das linke SQGM in Abbildung 5.9 in das rechte Modell derselben Abbildung.*

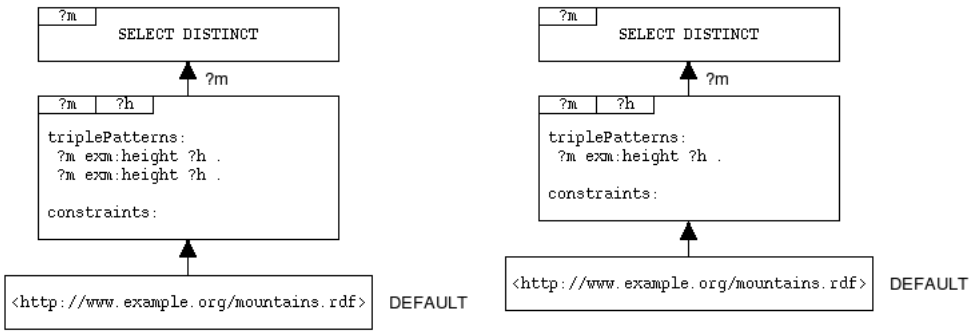


Abbildung 5.9: Ein SQGM vor und nach der Ausführung einer RemoveEqualTriplePatterns-Regelinstanz □

5.4.1 Anforderungen an das Regelausführungssystem

Ein System zur Ausführung der, im Rahmen dieser Arbeit entwickelten Transformationsregeln muss es der Anfragebearbeitungskomponente ermöglichen, eine Menge von Regelinstanzen auszuführen, um ein spezielles Ziel zu erreichen. Ein solches Ziel kann beispielsweise die Normalisierung eines SQGM sein. Diese Hauptaufgabe hat im Zusammenhang mit einer Untersuchung der im Kapitel 4 spezifizierten Transformationsregeln zu den folgenden Anforderungen geführt.

- Das Regelsystem muss eine Datenstruktur zur Verwaltung der Regelinstanzen besitzen. Diese Struktur muss eine Abarbeitungsliste beinhalten, um die Ausführungsreihenfolge der verwalteten Regelinstanzen festzulegen.
- Das Regelsystem muss die jeweils als nächstes abzuarbeitende Regelinstanz aus der Verwaltungsstruktur entfernen, ihre Ausführbarkeit prüfen und im Falle eines positiven Ergebnisses ausführen.
- Es muss jederzeit möglich sein, der Abarbeitungsliste weitere Regelinstanzen hinzuzufügen. Die Initialisierung der Liste mit entsprechenden Startregeln wird durch das zu erreichende Ziel bestimmt.
- Es muss jederzeit möglich sein bestimmte Regelinstanzen aus der Abarbeitungsliste zu entfernen, ohne dass diese abgearbeitet wurden.
- Das Regelsystem muss die Verwaltung und Abarbeitung von sicher ausführbaren und möglicherweise ausführbaren Folgeregeln gewährleisten.
  Die Ausführung einzelner Regelinstanzen kann dazu führen, dass unter bestimmten Umständen die Abarbeitung weiterer Regelinstanzen ermöglicht wird. Beispielsweise sind in den Tabellen 4.1 (Seite 57) und 4.4 (Seite 70) entsprechende Abhängigkeiten für einige der Regeln aus Kapitel 4 aufgeführt. Das Regelsystem muss die Verwaltung und Abarbeitung entsprechender Folgeregeln ermöglichen. Hierbei ist zwischen zwei Arten von Folgeregeln zu unterscheiden. Einerseits ist es denkbar, dass durch die Ausführung einer Regelinstanz direkt die Voraussetzungen einer anderen Regelinstanz erfüllt werden. Diese Folgeregel wird als *sicher ausführbar* bezeichnet und kann vom System ausgeführt werden, ohne ihre Voraussetzungen explizit überprüfen zu müssen. Ein Beispiel hierfür ist die Abhängigkeit zwischen den Regeln FindContradiction und HandleContrInput (siehe erste Zeile der Tabelle 4.4). Andererseits könnte eine Regelausführung lediglich die Möglichkeit zur Abarbeitung einer anderen Regelinstanz eröffnen. Diese Folgeregel müsste dann noch auf ihre tatsächliche Ausführbarkeit hin getestet werden. Sie wird als *möglicherweise ausführbar* bezeichnet.
- Das Hinzufügen möglicher Folgeregeln in die Abarbeitungsliste könnte dazu führen, dass mehrmals die gleiche Transformationregel für denselben Operator aufgenommen wird. Dieser Fall muss vermieden werden.
- Die Datenstruktur zur Verwaltung der Regelinstanzen muss die Möglichkeit besitzen, alle Regelinstanzen, die einen bestimmten Operator betreffen, aus der Abarbeitungsliste zu entfernen.
  Es gibt Transformationsregeln, die Operatoren aus dem SQGM entfernen. Beispielsweise entfernt die Regel RemoveEmptyInput (siehe Seite 59) einen leeren Graphmuster-Operator und die Regel HandleContrJoinInputLeft (Seite 66) entfernt sogar einen gesamten Teilbaum des SQGM. In diesen Fällen bewirken die abzuarbeitenden Regelinstanzen, welche die entfernten Operatoren betreffen, keine Veränderung des SQGM und können ebenfalls entfernt werden.

5.4.2 Entwurfsmodell des Regelausführungssystems

Unter Beachtung der aufgezählten Anforderungen wurde das, im Folgenden vorgestellte System zur Regelausführung entworfen. Hierfür wurde zunächst entschieden, jede der im Rahmen dieser

Arbeit definierten Transformationsregeln durch eine Klasse zu modellieren. Eine Regelinstanz wird dann durch ein Objekt der entsprechenden Klasse repräsentiert. Um die Gemeinsamkeiten aller Regeln zu vereinen und dem Regelausführungssystem eine einheitliche Schnittstelle zur Verwaltung und Ausführung der Regeln anzubieten, wurde die abstrakte Basisklasse `Rule` modelliert. Die beiden Hauptaufgaben des Regelausführungssystems, die Verwaltung und die Ausführung der Regelinstanzen, wurde getrennt voneinander durch die Klassen `RuleList` und `RuleEngine` modelliert. Diese Aufspaltung ermöglicht eine genaue Unterscheidung der beiden Funktionalitäten während der Implementierung und Wartung des Regelsystems. Das Klassendiagramm in Abbildung 5.10 zeigt die drei Klassen und deren Beziehungen und repräsentiert somit das entworfene System.

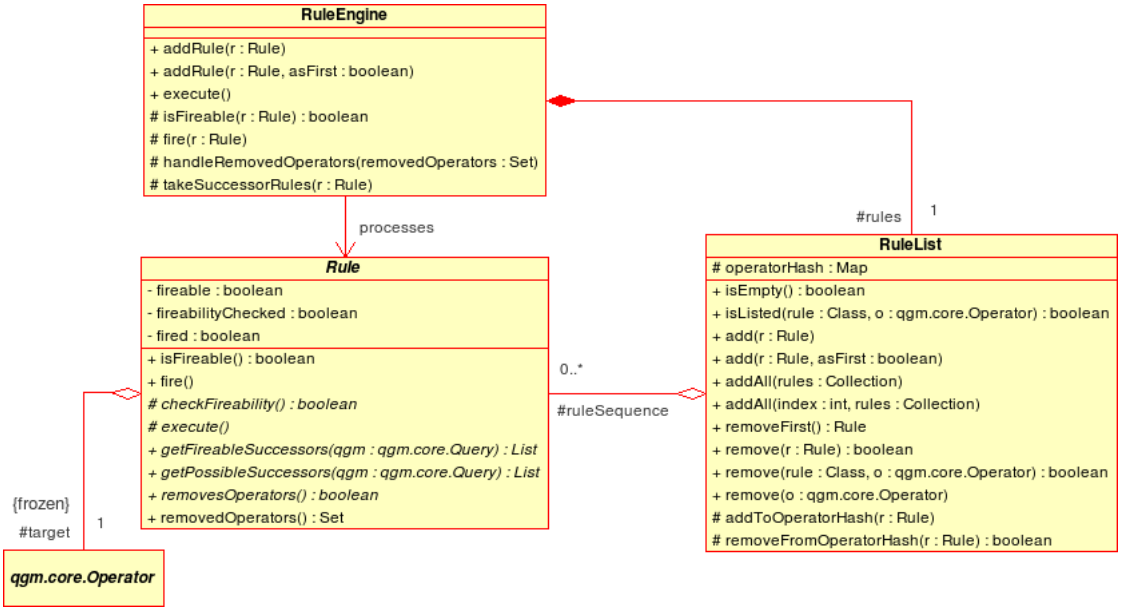


Abbildung 5.10: Entwurfsmodell der Klassen des Regelausführungssystems

Wie bereits erwähnt, enthält die abstrakte Basisklasse `Rule` die gemeinsame Funktionalität aller Transformationsregeln. Zusätzlich spezifiziert sie Operationen, die in den konkreten Regeln, entsprechend der speziellen Eigenschaften, implementiert werden müssen. Jede Regelinstanz betrifft genau einen Operator, welcher durch die dargestellte Komposition mit dem Attribut `target` modelliert wurde.

Die beiden Operationen `isFireable` und `fire` bieten dem Regelsystem die Möglichkeit zur Ausführbarkeitsüberprüfung und zur Ausführung jeder Regelinstanz. Beide Operationen kapseln die Aufrufe der tatsächlichen Funktionalität, welche in den speziellen Regelklassen durch die Implementierung der Operationen `checkFireability` und `execute` definiert werden muss.

Über die abstrakten Operationen `getFireableSuccessors` und `getPossibleSuccessors` wird dem Regelausführungssystem eine Schnittstelle zur Bestimmung von sicher ausführbaren und möglicherweise ausführbaren Folgeregeln angeboten und die Operationen `removesOperators` und `removedOperators` ermöglichen eine Reaktion auf die Entfernung von SQGM-Operatoren während der Regelausführung.

Die Klasse `RuleList` modelliert die Struktur zur Verwaltung der Regelinstanzen. Diese sind in dem durch die Aggregation dargestellten Attribut `ruleSequence` enthalten, welches als verkettete Liste implementiert wurde und die Abarbeitungsreihenfolge festlegt. Über die modellierten Operationen können Regelinstanzen hinzugefügt und entfernt werden. Hervorzuheben ist hier die Operation `removeFirst`, welche die nächste abzuarbeitende Regelinstanz aus der Liste entfernt und der Ausführungskomponente zur Verfügung stellt.

Alle abzuarbeitenden Regelinstanzen werden zusätzlich noch in einer Hashtabelle verwaltet, die durch das Attribut `operatorHash` repräsentiert wird. Diese Tabelle ordnet jedem Operator eine

Menge von, ihn betreffenden Regelinstanzen zu. Das ermöglicht die Implementierung der Forderung nach einer Möglichkeit zur Entfernung aller Regelinstanzen für einen bestimmten Operator. Hierfür dient die entsprechende `remove`-Operation.

Die Operationen zur Einordnung neuer Regelinstanzen in die Abarbeitungsliste bieten verschiedene Alternativen für deren Positionierung. Unter Ausnutzung dieser Möglichkeiten kann das Regelsystem unterschiedliche Abarbeitungsmodelle wie *Warteschlangen* (engl.: queue) oder *Stapel* (engl.: stack) implementieren.

Die Ausführungskomponente des entworfenen Regelausführungssystems wurde durch die Klasse `RuleEngine` modelliert. Sie verwaltet die abzuarbeitenden Regelinstanzen in einem `RuleList`-Objekt. Über die `addRule`-Operationen können abzuarbeitende Regelinstanzen spezifiziert werden. Dadurch kann die Anfragebearbeitungskomponente das Regelsystem mit einer Sequenz von Startregeln initialisieren. Durch die Operation `execute` wird die Abarbeitung der Regeln gestartet. Der Algorithmus in Abbildung 5.11 zeigt das Vorgehen bei der Regelabarbeitung.

```
ALGORITHMUS: PROCESSRULES
EINGABE: rules – Liste mit abzuarbeitenden Regelinstanzen (Objekte der Klasse RuleList)

while ( rules ist nicht leer ) do
        r := rules.removeFirst();
        if ( r ist ausführbar ) then
                Führe r aus;
                if ( r hat Operatoren entfernt ) then
                        Entferne alle Regelinstanzen zu den entfernten Operatoren aus rules;
                Übernimm Folgeregeln von r in rules;
```

Abbildung 5.11: Algorithmus zur Abarbeitung einer Menge von Regelinstanzen

Die einzelnen Aktivitäten des dargestellten Algorithmus PROCESSRULES wurden durch Operationen modelliert, welche von der Operation, die PROCESSRULES implementiert, genutzt werden. Dieses Vorgehen ermöglicht eine Erweiterung der Abarbeitung durch eine speziellere, von `RuleEngine` abgeleitete Ausführungskomponente. Die Standardimplementierungen der Operationen `isFireable` und `fire` rufen lediglich die entsprechenden Operationen der angegebenen Regelinstanz auf. In `handleRemovedOperators` wird einfach für jeden, der übergebenen Operatoren, die `remove`-Operation des `RuleList`-Objekts gerufen.

Der wahrscheinlichste Kandidat für eine Redefinition im Rahmen einer spezialisierten Ausführungskomponente ist die Operation `takeSuccessorRules`. Die Variante in der Standardimplementierung ist in der Abbildung 5.12 als Pseudocode dargestellt. Sie ordnet alle mit Sicherheit ausführbaren Folgeregeln einer Stapelabarbeitung entsprechend an den Anfang der Abarbeitungsliste. Alle möglicherweise ausführbaren Folgeregeln werden, einer Warteliste entsprechend, an das Ende der Abarbeitungsliste gehängt. Durch diesen hybriden Ansatz bezüglich des implementierten Abarbeitungsmodells werden zunächst immer alle ausführbaren Regelinstanzen abgearbeitet, bevor die zu überprüfenden Regelinstanzen in Betracht gezogen werden.

```
ALGORITHMUS: TAKESUCCESSORRULES
EINGABE: rules – Liste mit abzuarbeitenden Regelinstanzen (Objekte der Klasse RuleList)
EINGABE: r – ausgeführte Regelinstanz (Objekt der Klasse Rule)

if ( r wurde nicht ausgeführt ) then
        ERROR!

f := r.getFireableSuccessors();
if ( f ist nicht leer ) then
        rules.addAll( 0, f );

p := r.getPossibleSuccessors();
if ( p ist nicht leer ) then
        rules.addAll( p );
```

Abbildung 5.12: Algorithmus zum Übernehmen von Folgeregeln in die Abarbeitungsliste

5.4.3 Implementierung der Normalisierung eines SQGM

Auf Basis des vorgestellten, generischen Regelausführungssystems wurde ein System zur Normalisierung eines SQGM entwickelt. Dieses System implementiert die in Abschnitt 4.2 (ab Seite 51) definierten Transformationsregeln als Spezialisierungen der Basisklasse `Rule`. Für all diese Klassen wird an dieser Stelle die Klasse `SplitConstraintRule` beispielhaft vorgestellt. Sie ist in Abbildung 5.13 dargestellt und implementiert die Regel SplitConstraint (siehe Definition 4.8 auf Seite 58), welche eine Wertebeschränkung c eines Graphmuster-Operators o in zwei Wertebeschränkungen c_1 und c_2 aufspaltet, wenn der logische Ausdruck von c eine Konjunktion ist.

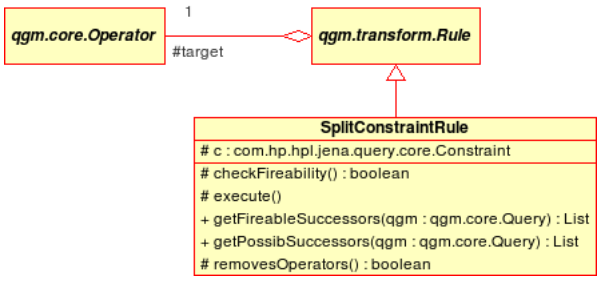


Abbildung 5.13: Entwurfsmodell der Regel SplitConstraint

Das Attribut `c` der Klasse `SplitConstraintRule` repräsentiert die aufzuspaltende Wertebeschränkung, falls eine solche existiert. Der betrachtete Graphmuster-Operator o wird, wie bereits erwähnt, durch das Kompositionsattribut `target` der Basisklasse identifiziert.

Die Operation `checkFireability` muss, entsprechend der Schnittstellendefinition, die Ausführbarkeit der Regel in Bezug auf o überprüfen. Hierzu wird über alle Wertebeschränkungen in o iteriert. Wird hierbei eine Wertebeschränkungen mit einer Konjunktion als logischem Ausdruck gefunden, so wird diese dem Attribut `c` zugewiesen und die Regelinstanz durch Rückgabe des Wahrheitswerts T als ausführbar gemeldet. Enthält o keine entsprechende Wertebeschränkung, so ist die Regelinstanz nicht ausführbar und es wird der Wahrheitswert F zurückgegeben.

In der `SplitConstraintRule`-Implementierung der Operation `execute` wird die Wertebeschränkung c aus der entsprechenden Liste des Operators o entfernt, zwei neue Wertebeschränkungen c_1 und c_2 aus c erzeugt und diese der Liste von o hinzugefügt. Da während der Ausführung kein Operator aus dem SQGM entfernt wird, gibt die implementierte Operation `removesOperators` den Wahrheitswert F zurück.

Für die Regel SplitConstraint sind keine sicher ausführbaren Folgeregeln bekannt. Deshalb gibt die entsprechende Implementierung der Operation `getFireableSuccessors` eine leere Liste zurück. Laut der Tabelle 4.2 (Seite 59) könnte die Regel für den betroffenen Operator noch einmal ausführbar sein. Demzufolge wird in der Operation `getPossibleSuccessors` eine neue SplitConstraint-Instanz bezüglich des betroffenen Operators erzeugt und der zurückgegebenen Liste hinzugefügt.

Das Klassendiagramm in Abbildung 5.14 stellt alle Klassen des Regelsystems zur SQGM-Normalisierung dar, wobei von den Transformationsregeln zur Normalisierung (siehe Abschnitt 4.2.3 ab Seite 56) lediglich die Regel SplitConstraint wie oben beschrieben modelliert und implementiert wurde.

Zusätzlich zu den bereits erwähnten Regel-Klassen beinhaltet das System die spezialisierte Ausführungskomponente `QGMNormalizer`. Sie erweitert die generische Basisklasse unter anderem um die Operation `removeEmptyOperators`, welche den in Abschnitt 4.2.4 (ab Seite 59) skizzierten Algorithmus zur rekursiven Entfernung von leeren Graphmuster-Operatoren implementiert. Eine zusätzliche Spezialisierung ist die Redefinition der Operation `execute`. Diese initialisiert die Abarbeitungsliste mit möglichen Startregeln, arbeitet die Regeln durch den Aufruf der Basisklassenfunktionalität ab und entfernt die leeren Graphmuster-Operatoren durch den Aufruf von `removeEmptyOperators`, wie der Pseudocode in Abbildung 5.15 verdeutlicht.

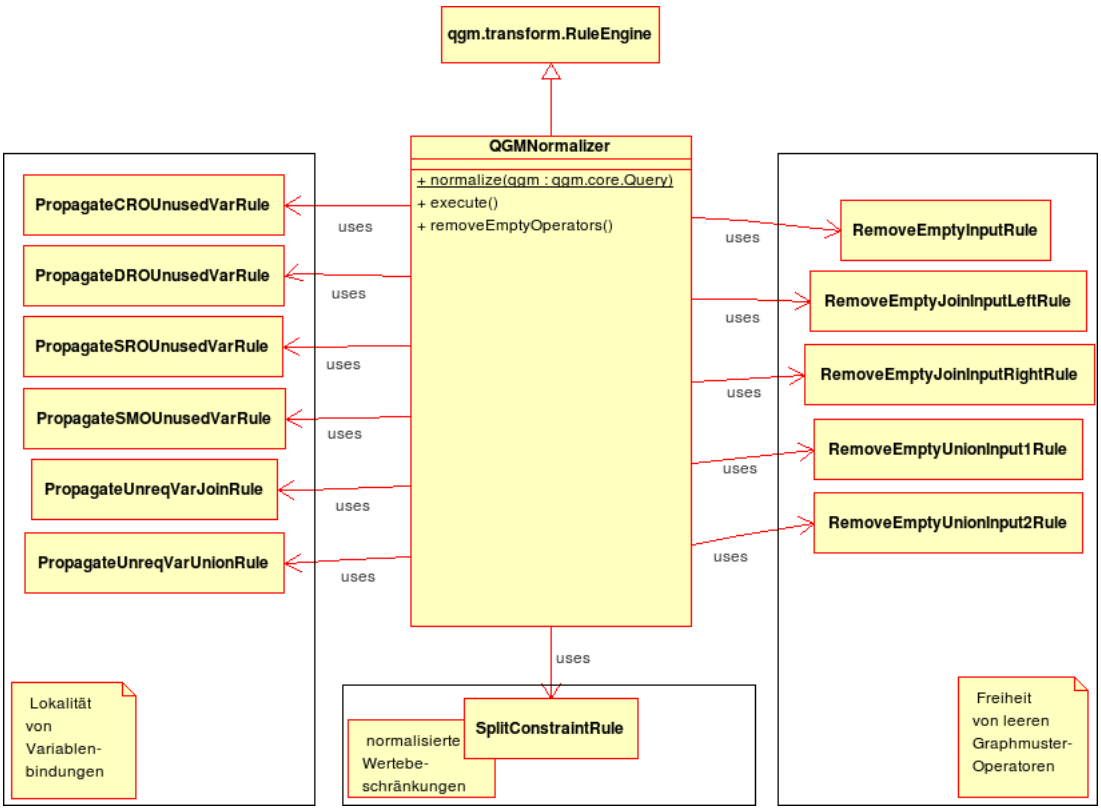


Abbildung 5.14: Entwurfsmodell der Regelsystems zur Normalisierung eines SQGM

ALGORITHMUS: EXECUTENORMALIZATION
EINGABE: $q := (OP, DF, r, dflt, NG)$ – das zu normalisierende SQGM (Objekt der Klasse `qgm.core.Query`)
AUSGABE: q' – das normalisierte SQGM

// Startregeln für Propagierung ungenutzter Variablenbindungen
Sei o_R der ergebnis-erzeugende Operator r von q;
`if` (o_R ist ein SELECT-Ergebnisoperator) `then`
    Erzeuge ein `PropagateSROUnusedVarRule`-Objekt r für o_R;
    addRule(r);
`elsif` (o_R ist ein DESCRIBE-Ergebnisoperator) `then`
    Erzeuge ein `PropagateDROUnusedVarRule`-Objekt r für o_R;
    addRule(r);
`elsif` (o_R ist ein CONSTRUCT-Ergebnisoperator) `then`
    Erzeuge ein `PropagateCROUnusedVarRule`-Objekt r für o_R;
    addRule(r);

// Startregeln für normalisierte Wertebeschränkungen
`foreach` ($o \in GPO \subset OP$) `do`
    Erzeuge ein `SplitConstraintRule`-Objekt r für o;
    addRule(r);

    // hier zusätzliche Erzeugung von NORMALIZECONSTRAINT-Instanzen, wenn es eine entsprechende
    // Implementierung dieser Regel gibt

Arbeite Regelinstanzen ab; // Basisklassenfunktionalität

removeEmptyOperators();

Abbildung 5.15: Algorithmus zur Normalisierung eines SQGM unter Verwendung des Regelausführungssystems

5.4.4 Implementierung der Vereinfachung eines SQGM

Analog zu dem System zur Normalisierung eines SQGM, wurde ein System zur Vereinfachung eines normalisierten SQGM auf Basis des generischen Regelausführungssystems prototypisch entwickelt. Die Abbildung 5.16 zeigt dessen Klassen. Es implementiert die in Abschnitt 4.3 (ab Seite 62) definierten Transformationsregeln und die spezialisierte Ausführungskomponente `QGMSimplifier`. Die Pseudocode-Darstellung in Abbildung 5.17 stellt den Algorithmus der redefinierten Operation `execute` von `QGMSimplifier` dar.

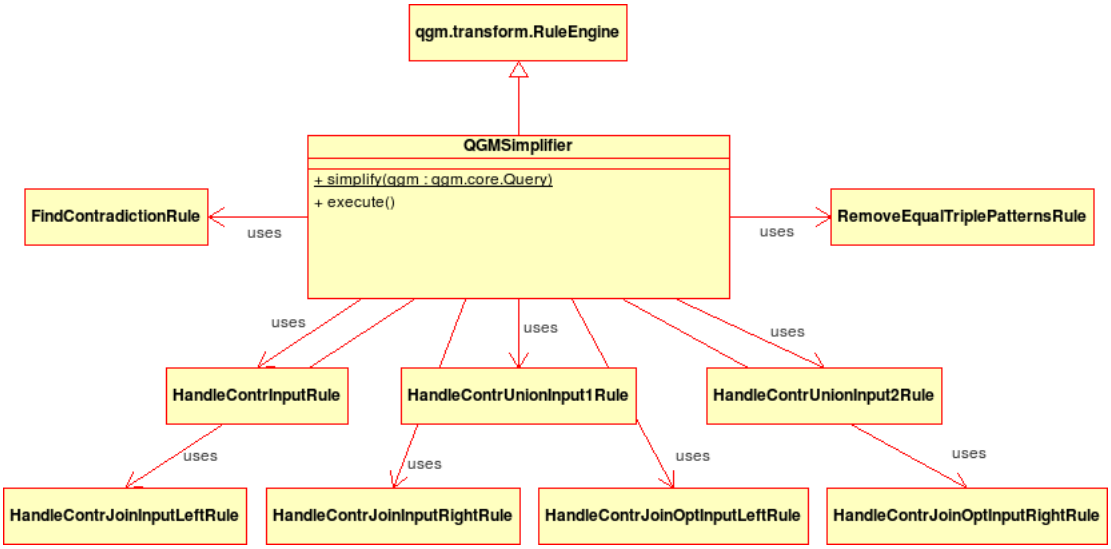


Abbildung 5.16: Entwurfsmodell der Regelsystems zur Vereinfachung eines normalisierten SQGM

ALGORITHMUS: EXECUTESIMPLIFICATION
EINGABE: $q := (OP, DF, r, dflt, NG)$ – das zu vereinfachende SQGM (Objekt der Klasse `qgm.core.Query`)
AUSGABE: q' – das vereinfachte SQGM

// Startregeln zur Entfernung von Redundanzen und zur Erkennung/Behandlung von Widersprüchen
`foreach` ($o \in GPO \subset OP$) `do`
    Erzeuge ein `RemoveEqualTriplePatternsRule`-Objekt r_1 für o;
    addRule(r_1);
    Erzeuge ein `FindContradictionRule`-Objekt r_2 für o;
    addRule(r_2);

Arbeite Regelinstanzen ab; // Basisklassenfunktionalität

Abbildung 5.17: Algorithmus zur Vereinfachung eines SQGM unter Verwendung des Regelausführungssystems

5.5 Implementierung der regelbasierten Restrukturierung

Die Ausführung der Instanz einer Transformationsregel verändert laut Definition 5.1 (Seite 79) in einem SQGM lediglich den betroffenen Operator oder die direkt mit ihm verbundenen Operatoren. Auf Basis dieser Transformationsregeln müssen Restrukturierungsregeln entwickelt werden, welche ein gesamtes SQGM mit einem speziellen Ziel restrukturieren. Diese Ziele und damit die konkreten Restrukturierungsregeln sind die Grundlage der untersuchten Heuristiken (siehe Kapitel 6 ab Seite 90).

In diesem Abschnitt wird die Entwicklung einer Komponente zur Restrukturierung eines SQGM beschrieben. Dazu wird zunächst die Bedeutung des Begriffs *Restrukturierungsregel* im Rahmen

dieser Arbeit spezifiziert. Danach werden die Anforderungen an die Restrukturierungskomponente formuliert und abschließend die entworfenen Klassen vorgestellt.

Definition 5.5 Eine *Restrukturierungsregel* (engl.: rewrite rule) wandelt ein SQGM q in ein semantisch äquivalentes SQGM q' um, wobei ein oder mehrere Transformationsregelinstanzen ausgeführt werden. Die Umwandlung verfolgt ein definiertes Ziel, welches durch mehrere Schritte der Restrukturierung des SQGM erreicht wird. Jeder dieser Schritte kann durch die Abarbeitung einer Sequenz von Transformationsregelinstanzen und möglicher Folgeregeln ausgeführt werden. Das Ziel gilt als erreicht, wenn keine weiteren Schritte ausführbar sind. Die Definition einer konkreten Restrukturierungsregel beinhaltet

- einen Namen,
- die Spezifikation des Ziels,
- die Spezifikation der Restrukturierungsschritte zur Erreichung des Ziels,
- einen Algorithmus zur Feststellung, ob ein weiterer Schritt ausführbar ist und
- einen Algorithmus zur Erstellung der initialen Sequenz von Transformationsregelinstanzen für einen ausführbaren Restrukturierungsschritt. □

Für konkrete Beispiele von Restrukturierungsregeln sei an dieser Stelle lediglich auf die Definitionen 6.1 (Seite 95) und 6.2 (Seite 117) im folgenden Kapitel verwiesen.

5.5.1 Anforderungen an die Komponente zur Restrukturierung

Eine Komponente zur Restrukturierung eines SQGM hat die Aufgabe die Restrukturierungsschritte einer oder mehrerer Restrukturierungsregeln auszuführen bis die Ziele aller Regeln erreicht sind. Hierzu muss das System folgende Anforderungen erfüllen.

- Der Zugriff auf die Implementierung der konkreten Restrukturierungsregeln muss über eine einheitliche Schnittstelle möglich sein. Insbesondere müssen die jeweiligen Algorithmen zur Feststellung der Ausführbarkeit eines weiteren Schrittes und zur Erstellung einer initialen Sequenz einheitlich aufgerufen werden können.
- Solange das Ziel von mindestens einer Restrukturierungsregel nicht erfüllt ist, muss eine der Regeln, für die ein weiterer Schritt ausführbar ist, ausgewählt und der entsprechende Schritt abgearbeitet werden.
- Es muss beachtet werden, dass sich für eine Regel, deren Ziel bereits erreicht ist, die Möglichkeit für die Ausführung eines weiteren Schrittes eröffnet, wodurch die Erreichung des Ziels aufgehoben ist. Das könnte der Fall sein, wenn die Ausführung eines Schrittes einer anderen Restrukturierungsregel zwischenzeitlich das betrachtete SQGM verändert hat.
- Die Reihenfolge, in der die einzelnen Restrukturierungsregeln für die Auswahl in Betracht gezogen werden, muss in der Restrukturierungskomponente exakt definiert sein.
- Die zur Ausführung eines Schrittes notwendige Sequenz von Transformationsregelinstanzen und möglicher Folgeregeln muss mit Hilfe eines Regelausführungssystems (siehe Abschnitt 5.4 ab Seite 78) abgearbeitet werden.

An dieser Stelle muss darauf hingewiesen werden, dass die Restrukturierungskomponente nicht überprüfen kann, ob die Abarbeitung der ihr zugewiesenen Restrukturierungsregeln terminiert. Es könnte sein, dass sich deren Ziele widersprechen, was unter Umständen während der Abarbeitung zu einer endlosen, abwechselnden Ausführung derselben Schritte führt. Dieses mögliche Problem ist bei der Festlegung der konkreten Regeln, welche von der Komponente abgearbeitet werden sollen, dringend zu beachten und zu vermeiden.

5.5.2 Entwurfsmodell des Restrukturierungssystems

Auf Grundlage der spezifizierten Anforderungen wurde eine Restrukturierungskomponente entwickelt. Das Klassendiagramm in Abbildung 5.18 stellt die dafür entworfenen Klassen und deren Beziehungen dar.

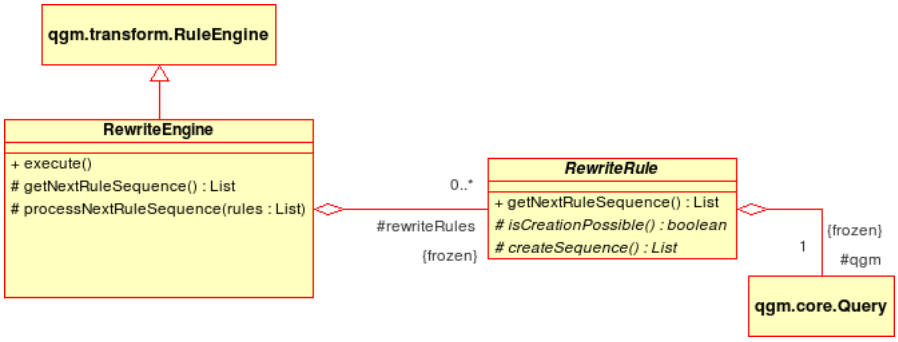


Abbildung 5.18: Entwurfsmodell der Klassen des Restrukturierungssystems

Die konkreten Restrukturierungsregeln müssen jeweils durch eine Klasse implementiert werden. Um die Anforderung nach einer einheitlichen Schnittstelle zu erfüllen, müssen diese Klassen von der entworfenen Basisklasse `RewriteRule` abgeleitet sein. Ihre Operation `getNextRuleSequence` bietet der Restrukturierungskomponente den gewünschten, standardisierten Zugriff. Hierfür kapselt die Implementierung dieser Operation den Aufruf der tatsächlichen Funktionalität, welche in den speziellen Regelklassen durch die Implementierung der Operationen `isCreationPossible` und `createSequence` definiert werden muss.

Die Klasse `RewriteEngine` repräsentiert die Komponente zur Restrukturierung eines SQGM. Sie besitzt die Liste `rewriteRules` von konkreten Restrukturierungsregeln, welche für die Restrukturierung verwendet werden. Über den Inhalt dieser Liste, welche der Restrukturierungskomponente während der Initialisierung übergeben wird, kann die eingesetzte Heuristik festgelegt werden. Wie bereits erwähnt ist bei der Festlegung der konkreten Regeln, welche in die jeweilige Liste aufgenommen werden, darauf zu achten, dass sich deren Ziele nicht widersprechen.

`RewriteEngine` ist eine Spezialisierung der Klasse `RuleEngine`, welche die Ausführungskomponente des entworfenen Regelausführungssystems (siehe Abschnitt 5.4.2 ab Seite 80) repräsentiert. Dadurch kann die jeweils zur Ausführung eines Restrukturierungsschrittes notwendige Sequenz von Transformationsregelinstanzen und möglicher Folgeregeln einfach durch die Verwendung der Basisklassenfunktionalität abgearbeitet werden. Der Algorithmus in Abbildung 5.19 zeigt die Abarbeitung der Restrukturierungsregeln. Er wurde durch die Operation `execute` implementiert.

ALGORITHMUS: PROCESSREWRITERULES
EINGABE: *rewriteRules* – Liste mit Restrukturierungsregeln (Objekte der Klasse `RewriteRule`)

finished := F;
`while` (*finished* ≠ T) `do`
    rules := nächste abzuarbeitenden Sequenz von einer Regel aus *rewriteRules*;
    `if` (*rules* ist definiert und nicht leer) `then`
        Füge alle Elemente aus *rules* zur Liste der abzuarbeitenden Regelinstanzen hinzu;
        Arbeite Regelinstanzen ab; // Basisklassenfunktionalität
    `else`
        finished := T;

Abbildung 5.19: Algorithmus zur Abarbeitung einer Menge von Restrukturierungsregeln

Der implementierte Algorithmus verwendet für die Bestimmung der nächsten abzuarbeitenden Sequenz von Transformationsregelinstanzen die Operation `getNextRuleSequence`. Diese benutzt

für die Auswahl der, von den einzelnen Restrukturierungsregeln angebotenen Sequenzen, die Reihenfolge der Regeln in der Liste `rewriteRules`. Demzufolge ist die Priorität des jeweils ausführbaren Schrittes jeder Regel von der Position der Regel in der Liste abhängig und die ausführbaren Schritte der ersten Regel haben immer Vorrang. Auf diese Weise wird die Forderung nach einer exakt definierten Reihenfolge der einzelnen Restrukturierungsregeln erfüllt.

Die hier vorgestellte Restrukturierungskomponente bildet lediglich die Grundlage für eine regelbasierte Restrukturierung. Um ein SQGM tatsächlich bezüglich eines oder mehrerer Ziele zu restrukturieren, ist die Definition und Implementierung konkreter Restrukturierungsregeln notwendig. In Kapitel 6 werden mehrere solcher Regeln vorgestellt und diskutiert.

5.6 Implementierung einer Testumgebung

Eine der Aufgaben der regelbasierten Optimierung ist die Anfragerestrukturierung auf Basis von Heuristiken mit dem Ziel einer effizienteren Anfrageausführung (siehe Abschnitt 3.3.2 ab Seite 35). Um mögliche Heuristiken bezüglich der Verkürzung von Ausführungszeiten untersuchen und beurteilen zu können, ist es notwendig diese Zeiten zu bestimmen. Hierfür wurde eine Testumgebung auf Basis der in den vorherigen Abschnitten vorgestellten Implementierung entwickelt.

Diese Testumgebung wird hier kurz vorgestellt. Sie bietet neben der Erfassung von Anfragebearbeitungszeiten noch die Möglichkeit zur Ausführung von Regressionstests. Die wichtigsten hierfür entworfenen Klassen und deren Beziehungen sind in der Abbildung 5.20 dargestellt.

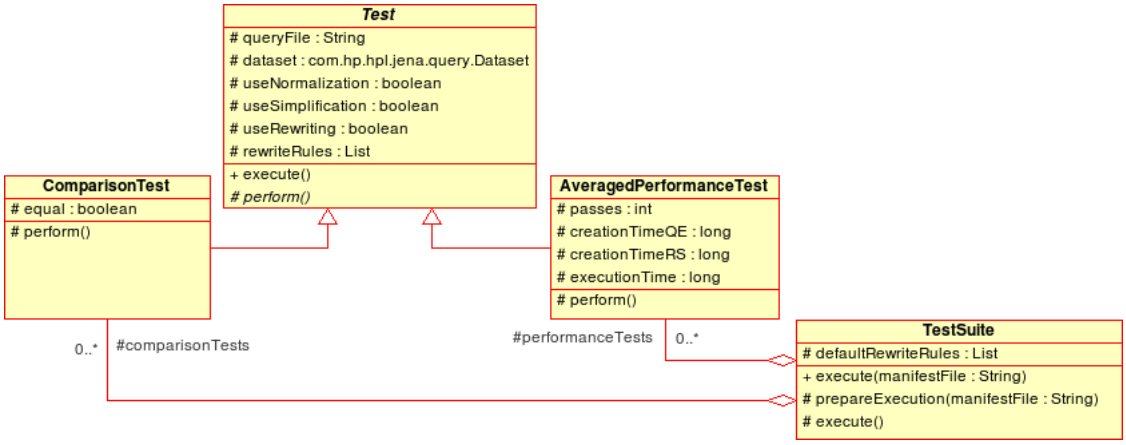


Abbildung 5.20: Entwurfsmodell der wichtigsten Klassen für die Testumgebung

Im Rahmen der entwickelten Testumgebung wird unter einem Testfall die Untersuchung der Ausführung von genau einer SPARQL-Anfrage über einer RDF-Datenmenge mittels der prototypischen Anfragebearbeitungskomponente `QGMTestEngine` verstanden. Die entworfene Basisklasse `Test` repräsentiert einen solchen Testfall. Die tatsächliche Art der Untersuchung der Anfrageausführung ist von der konkreten, von `Test` abgeleiteten Klasse abhängig. Die Eigenschaften mit denen die Anfragebearbeitungskomponente für diese spezielle Ausführung initialisiert wird, sind durch die Attribute `useNormalization`, `useSimplification` und `rewriteRules` der Klasse `Test` spezifiziert, wobei insbesondere auf die Liste `rewriteRules` hingewiesen sei. Diese Liste enthält die Restrukturierungsregeln mit denen die von der Anfragebearbeitungskomponente eingesetzte Restrukturierungskomponente initialisiert wird.

Die Klasse `ComparisonTest` repräsentiert einen Testfall, bei dem die Korrektheit der Anfrageausführung untersucht werden kann. Hierfür wird die Anfrage neben der prototypischen Anfragebearbeitungskomponente zusätzlich mit der Standard-Komponente von ARQ ausgeführt und

die erhaltenen Ergebnisse miteinander verglichen. Dem Attribut `equal` wird nach der Ausführung das Resultat des Vergleichs zugewiesen. Mit Hilfe dieser Klasse können Regressionstests für verschiedene Restrukturierungsregeln und Kombinationen dieser durchgeführt werden.

Die bereits erwähnte Untersuchung der Anfragebearbeitungszeit wird durch die Testfall-Klasse `AveragedPerformanceTest` unterstützt. Bei der Abarbeitung wird der Testfall n-mal durchgeführt. Die konkrete Anzahl der Wiederholungen ist durch das Attribut `passes` festgelegt. In jedem Schritt werden die Zeiten

- zur Initialisierung der Anfragebearbeitungskomponente,
- zur Erstellung des Ausführungsplans und
- zur Ausführung

gemessen und am Ende entsprechende Mittelwerte berechnet. Die drei Mittelwerte werden als Messwerte der *Initialisierungszeit*, der *Planerzeugungszeit* und der *Anfrageausführungszeit* übernommen und den Attributen `creationTimeQE`, `creationTimeRS` und `executionTime` zugewiesen. Aus diesen drei Messwerten können weitere Eigenschaften der speziellen Ausführung bestimmt werden. Zum Beispiel entspricht die Summe aller drei Werte der gesamten *Anfragebearbeitungszeit* und die Summe der ersten beiden Werte ergibt die *Ausführungsvorbereitungszeit*.

Bei der prototypischen Anfragebearbeitungskomponente `QGMTestEngine` umfasst die Initialisierung auch die Erzeugung und Restrukturierung des SQGM (siehe Abbildung 5.8 auf Seite 78). Dadurch kann durch die Messwerte auch das Verhältnis der Zeit für die Anwendung der, in dieser Arbeit vorgestellten Algorithmen zur Zeit für die Anfrageausführung bestimmt werden.

Die Klasse `TestSuite` repräsentiert eine Sammlung von Testfällen, welche hintereinander ausgeführt werden können. Die konkreten Testfälle müssen mittels eines RDF-Dokumentes beschrieben werden. Die Operation `prepareExecution` der Klasse `TestSuite` erzeugt anhand des Dokumentes die entsprechend initialisierten `ComparisonTest`- und `AveragedPerformanceTest`-Objekte.

Das für die RDF-Dokumente entwickelte Vokabular erlaubt die Aufzählung einer Liste von Testfällen. Jede Testfallbeschreibung besteht aus der Angabe einer Datei, welche die entsprechende SPARQL-Anfrage enthält, und optionalen Angaben zu der verwendeten Anfragebearbeitungskomponente. Diese optionalen Angaben können festlegen, dass bei der Ausführung des entsprechenden Testfalls die Normalisierung, die Vereinfachung oder die Restrukturierung ausgelassen werden sollen. Zusätzlich kann zu jedem Testfall eine Liste von zu verwendenden Restrukturierungsregeln spezifiziert werden. Für alle Testfälle bei denen keine spezielle Liste angegeben ist, erlaubt das entwickelte RDF-Vokabular die Angabe einer standardmässig anzuwendenden Liste von Restrukturierungsregeln. Auf diese Art ist es möglich verschiedenste Testszenarien zu entwickeln und in einem Durchgang abarbeiten zu lassen.

Kapitel 6

Heuristiken für regelbasierte Optimierung

Im Rahmen der Diskussion der Anfragekompilierung auf Basis des SQGM wurden eine effizientere Anfrageausführung und die Erhöhung der Möglichkeiten zur Planerzeugung als Ziele der regelbasierten Optimierung formuliert (siehe Abschnitt 3.3.2 auf Seite 35). Inbesondere die Erreichung einer effizienteren Anfrageausführung basiert in diesem Zusammenhang auf der Anwendung von Heuristiken.

Dieses Kapitel widmet sich solchen Heuristiken, wobei der erste Abschnitt die wichtigsten Eigenschaften von Heuristiken benennt und beschreibt. Der zweite Abschnitt stellt die eingesetzte Testumgebung zur Untersuchung von implementierten Heuristiken vor und in den folgenden Abschnitten werden einige Heuristiken für die regelbasierte Optimierung auf Basis eines SQGM diskutiert. Hierzu werden die jeweiligen Heuristiken zunächst motiviert und die Vermutung für ihre Einsetzbarkeit erläutert. Daraufhin werden die notwendigen Transformations- und Restrukturierungsregeln spezifiziert und abschließend werden diese auf Basis der, in Kapitel 5 vorgestellten SQGM-Implementierung in der Testumgebung experimentell untersucht.

6.1 Grundlegende Eigenschaften von Heuristiken

Der Begriff Heuristik ist von dem griechischen Wort *heurískein* abgeleitet, welches im Deutschen „finden" oder „entdecken" bedeutet. Nach [Sch98] ist eine Heuristik eine Methode „die bei der Bearbeitung komplexer Probleme mit Hoffnung auf, aber ohne Garantie von Erfolg eingesetzt wird" und laut [Mey01] bezeichnet man als Heuristik „ein Lösungsverfahren, das nur zum Teil auf wissenschaftlich gesicherten Erkenntnissen, sondern vorwiegend auf Hypothesen, Analogien und Erfahrungen aufbaut. Die Güte solcher Verfahren ist deshalb meist nicht beweisbar, sondern wird durch wiederholte Experimente an typischen Problemstellungen nachgewiesen".

Im Rahmen der regelbasierten Optimierung dienen Heuristiken der Verbesserung eines SQGM für eine effizientere Anfrageausführung. Sie basieren auf Hypothesen und Vermutungen und gelten in den meisten Fällen, sind jedoch nicht allgemeingültig. In dieser Arbeit werden Heuristiken durch

- die Hypothese,
- eine bestimmte Menge an Voraussetzungen,
- ein Effizienzkriterium und
- eine oder mehrere Restrukturierungsregeln

spezifiziert. Für die angegebenen Voraussetzungen gilt dann, dass die Restrukturierung eines SQGM durch Anwendung der festgelegten Restrukturierungsregeln in den meisten Fällen zu einer, bezüglich des Kritriums, effizienteren Ausführung der Anfrage führt.

Mögliche Voraussetzungen können sich auf die Art der Anfrage, die Art, Menge und Speicherform der Daten oder das eingesetzte System zur Anfrageausführung beziehen. Die Anfragen könnten in diesem Zusammenhang nach ihrer Größe, den verwendeten Klauseln oder der Kohärenz zwischen den enthaltenen Tripelmustern klassifiziert werden. Entsprechend ist eine Unterscheidung der Art der abgefragten Daten nach der Zusammengehörigkeit der enthaltenen Tripel und möglicher Tripel-Cluster, aber auch nach tatsächlich getroffenen oder durch Inferenz geschlossenen Aussagen möglich. Zusätzlich kann die Verwendbarkeit von Schemadaten eine Rolle spielen. Als Speicherformen für die Daten kommen hauptsächlich in einer Datenbank oder im Hauptspeicher vorliegende oder auf verschiedene Netzwerkknoten verteilte RDF-Datenmengen in Frage.

Die Vielzahl und Heterogenität der möglichen Voraussetzungen erlauben die Formulierung und Untersuchung einer großen Anzahl verschiedenster Heuristiken, wobei viele der Voraussetzungen eine eingehendere Auseinandersetzung mit den genannten Eigenschaften und Anwendungsumgebungen erfordern. Zusätzlich gibt es verschiedene Kriterien für die Effizienz der Anfrageausführung. Um den Rahmen dieser Arbeit nicht zu sprengen, konzentrieren sich die im Folgenden beispielhaft diskutierten Heuristiken auf einfache Anfragen an persistent gespeicherte Datenmengen, welche durch die Anfragebearbeitungskomponente von ARQ (siehe Abschnitt 5.1 ab Seite 71) ausgeführt werden. Dabei wird lediglich die Ausführungszeit als Effizienzkriterium betrachtet.

6.2 Die Testumgebung

Um die folgenden Heuristiken bezüglich der Güte der zugrundeliegenden Vermutungen untersuchen zu können, wurde eine Testumgebung eingerichtet, welche in diesem Abschnitt vorgestellt wird.

Als Testrechner wurde ein PC mit einem AMD Athlon XP 2800+ Prozessor und 512 MB Hauptspeicher unter SuSE-Linux (v.10.1) mit der enthaltenen Java-Umgebung (v.1.4.2.11) von Sun eingesetzt. Als Jena-Framework wurde die mit dem ARQ-Paket (v.1.3) bereitgestellte Version verwendet. Der Java-Interpreter wurde jeweils mit einer initialen Speichergröße von 64 MB (`-Xms64m`) und einer maximalen Speichergröße von 256 MB (`-Xmx256m`) gestartet.

Zur Erzeugung der notwendigen Testdaten wurde der Generator UBA[1] (v.1.7) des Lehigh University Benchmarks [GPH05] eingesetzt, welcher auf einer Ontologie zur Beschreibung von Begriffen und deren Beziehungen aus dem Universitäts-Umfeld basiert. Der Generator erzeugt künstliche Individuen dieser spezifizierten Klassen und Beziehungen. Die Datenerzeugung ist zufalls-basiert aber durch die Festlegung eines Startwertes für den Zufallsprozessor eindeutig wiederholbar und die Größe der generierten Datenmenge ist durch die Angabe der Anzahl zu erzeugender Universitäts-Individuen beliebig skalierbar.

Für die Untersuchungen im Rahmen dieser Arbeit wurden drei Datenmengen unterschiedlicher Größe erzeugt, wobei der Zufallsprozessor des Daten-Generators jeweils mit dem Startwert 0 initialisiert wurde. Die kleinste Datenmenge wurde für ein Universitäts-Individuum erzeugt und wird im Folgenden als UnivBench(1,0) bezeichnet. Die erzeugte Menge enthält 100 543 Tripel, welche 20 659 verschiedene Ressourcen beschreiben. Die zweite Datenmenge, im Folgenden als UnivBench(5,0) bezeichnet, wurde für fünf Universitäts-Individuen erzeugt und enthält 624 532 Tripel über 129 533 Ressourcen. Mit 1 272 575 Tripeln ist UnivBench(10,1) die größte Datenmenge. Sie wurde für zehn Universitäts-Individuen generiert und beschreibt 263 427 verschiedene Ressourcen.

Um die Untersuchung von SPARQL-Anfragen über einer persistenten Datenmenge zu ermöglichen, wurde ein DBMS (MySQL v.5.0.18) aufgesetzt, mittels Jena drei Testdatenbanken eingerichtet und diese mit den Testdaten gefüllt. Daraufhin wurden für die Datenmengen verschiedenste SPARQL-Anfragen formuliert und diese zur experimentellen Untersuchung der folgenden Heuristiken mit Hilfe der in Abschnitt 5.6 (Seite 88) vorgestellten Testfall-Klassen ausgeführt.

[1] http://swat.cse.lehigh.edu/projects/lubm/

6.3 Ausnutzung von ARQ's Fast-Path-Algorithmus für geeignete, gruppierte Tripelmuster

In diesem Abschnitt wird eine Heuristik formuliert und untersucht, welche sich auf SPARQL-Anfragen bezieht die Graphmuster-Gruppen mit Tripelmustern enthalten und von der ARQ-Anfragebearbeitungskomponente über einer persistent gespeicherten Datenmenge ausgeführt werden.

6.3.1 Zugrundliegende Vermutung

Im Abschnitt 5.1 (ab Seite 71) wurde der Fast-Path-Algorithmus von Jena für RDF-Daten in Datenbanken erwähnt. Es wurde vermutet, dass, wie im Folgenden beschrieben, die Eigenschaften des Algorithmus für eine schnellere Anfrageausführung ausgenutzt werden können. Zunächst muss der Algorithmus jedoch näher erläutert werden, um die zugrundeliegende Vermutung nachvollziehen zu können.

Der Fast-Path-Algorithmus wird angewandt, um während der Anfrageausführung die Musterübereinstimmungen für einzelne einfache Graphmuster (siehe Abschnitt 2.3.3 ab Seite 10) mit Wertebeschränkungen zu finden. Hierzu unterteilt der Algorithmus die Tripelmuster des einfachen Graphmusters in verschiedene Gruppen. Diese Gruppen werden als *Stufen* (engl.: stages) der Lösung bezeichnet und jede Gruppe beinhaltet jeweils die Muster, die zusammen abgearbeitet werden können.

Zur Bestimmung von Musterübereinstimmungen für Gruppen mit nur einem Muster wird während der Anfrageausführung lediglich die `find`-Operation von Jenas `Graph`-Schnittstelle (siehe Abschnitt 5.1 ab Seite 71) aufgerufen. Deren Implementierung für persistent gespeicherte RDF-Datenmengen wählt zunächst eine SQL-*Anfrage-Schablone* (engl.: query template) für das entsprechende Tripelmuster aus, wobei diese Auswahl von der Art der Knoten im Tripelmuster bestimmt wird. Entsprechend der acht Möglichkeiten für das Vorkommen von Variablen innerhalb eines Tripelmusters kennt Jena acht verschiedene Schablonen. Nach der Auswahl wird die Schablone für das Tripelmuster initialisiert und mit Hilfe des DBMS ausgeführt.

Im Gegensatz dazu wird für jede Gruppe mit mehr als einem Muster jeweils eine kombinierte SQL-Anfrage dynamisch erzeugt. Diese Anfrage enthält Verbundoperationen zur Verknüpfung der Auswertung der einzelnen Tripelmuster der jeweiligen Gruppe. Dadurch wird ein Teil der Anfrageoptimierung dem DBMS übertragen und der Aufwand der Anfrageausführung innerhalb von ARQ verringert.

Für die Abarbeitung der Musterübereinstimmung des gesamten einfachen Graphmusters werden die einzelnen Lösungsstufen im Sinne einer strombasierte Verarbeitung miteinander verkettet. Demzufolge bilden die in der n-ten Stufe gefundenen Musterübereinstimmungen die Basis für die bereits gebundenen Variablen der $n + 1$-ten Stufe. Die Abarbeitung wird mit einer zusätzlichen Stufe abgeschlossen, in der die Ergebnismenge bezüglich der Wertebeschränkungen reduziert wird. Für spätere Jena-Versionen ist eine Erweiterung des Algorithmus angekündigt, bei der die Wertebeschränkungen möglichst auf die Gruppen verteilt und in die Datenbankanfragen integriert werden sollen.

Abschließend muss noch erläutert werden, nach welchen Kriterien die Gruppen gebildet werden. Diese Klassifikation ist hauptsächlich von dem Datenbank-Schema abhängig, welches von Jena zur persistenten Speicherung von RDF-Daten verwendet wird. Dieses Schema ist in [WSKR03] beschrieben. Demnach können derzeit grundsätzlich nur die Muster in einer Gruppe vereinigt werden, die sich auf dieselbe Datenbanktabelle beziehen und denen mindestens eine Variable gemeinsam ist. Als zusätzliche Einschränkung sind aufgrund des Datenbank-Schemas Tripelmuster mit einer Variable an der Position des Prädikats von der Gruppierung mit anderen Tripelmustern ausgeschlossen. Diese Muster bilden derzeit immer eine eigene Gruppe.

Die Hauptidee der Heuristik ist die Verschmelzung einzelner einfacher Graphmuster einer SPARQL-Anfrage mit dem Ziel, während der Abarbeitung des Fast-Path-Algorithmus umfangreichere Gruppen erzeugen zu können. Es wurde vermutet, dass sich der reduzierte Aufwand der Anfrageausführung innerhalb von ARQ und die geringere Anzahl an auszuführenden Datenbankanfragen positiv auf die Zeit zur Ausführung der SPARQL-Anfrage auswirken.

Beispiel 6.1 *Die SPARQL-Anfrage*

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n ?h ?a
WHERE {
   ?m exm:name ?n .
   {
      ?m exm:height ?h .
      OPTIONAL { ?m exm:firstAscent ?a }
   }
}
```

fragt jeweils nach dem Namen, der Region, der Höhe und optional nach der Erstbesteigung aller in der abgefragten Datenmenge enthaltenen Berge. Ihre WHERE-Klausel ist eine Graphmuster-Gruppe. Diese enthält ein einfaches Graphmuster mit einem Tripelmuster und eine weitere Graphmuster-Gruppe. In dieser sind ein weiteres einfaches Graphmuster und ein optionales Graphmuster gruppiert. Die beiden einfachen Graphmuster können verschmolzen und die zweite Graphmuster-Gruppe aufgelöst werden und es entsteht die folgende, semantisch äquivalente Anfrage.

```
PREFIX exm: <http://www.example.org/mountains/>
SELECT ?n ?h ?a
WHERE {
   ?m exm:name ?n .
   ?m exm:height ?h .
   OPTIONAL { ?m exm:firstAscent ?a }
}
```

Die Graphmuster-Gruppe dieser Anfrage enthält ein einfaches Graphmuster mit zwei Tripelmustern und das optionale Graphmuster. Bei einer Ausführung beider Anfragen durch die ARQ-Anfragebearbeitungskomponente über einer persistenten Datenmenge wird vermutet, dass die zweite Anfrage schneller ausgeführt werden kann, weil der Fast-Path-Algorithmus in diesem Fall die Musterübereinstimmung für beide Tripelmuster in einer Lösungsstufe verbinden kann. □

Diese Heuristik setzt natürlich voraus, dass eine Anfrage mehrere einfache Graphmuster enthält, welche unter Umständen in einer oder mehreren Graphmuster-Gruppen enthalten sind. Auch wenn im obigen Beispiel die erste, vermutlich ineffizientere Variante der Anfrage speziell für das Beispiel konstruiert wurde, ist davon auszugehen, dass diese oder ähnlich Fälle in der Praxis häufig auftreten. Insbesondere in generierten SPARQL-Anfragen sind entsprechende Muster-Gruppierungen zu erwarten.

Die Aufgabe bei der Anwendung der Heuristik besteht darin, die einzelnen einfachen Graphmuster innerhalb der Graphmuster-Gruppen auf ihre Verschmelzbarkeit hin zu überprüfen und im positiven Fall die Verschmelzung vorzunehmen. Die Prüfung der Verschmelzbarkeit dient zur Gewährleistung der semantischen Äquivalenz beider Anfragen. So kann ein einfaches Graphmuster innerhalb eines optionalen Graphmusters nicht mit einem anderen einfachen Graphmuster ausserhalb verschmolzen werden. Im Folgenden wird die Umsetzung der skizzierten Aufgabe für diese Heuristik auf Basis des SQGM beschrieben.

6.3.2 Notwendige Restrukturierungsregeln

In einem SQGM werden Graphmuster-Gruppen durch Verbund-Operatoren (siehe Definition 3.14 auf Seite 25) oder Bäume von Verbund-Operatoren und einfache Graphmuster durch Graphmuster-Operatoren (siehe Definition 3.12 auf Seite 24) modelliert. Demzufolge muss für die Heuristik eine

Restrukturierungsregel entwickelt werden, die möglichst alle verbundenen Graphmuster-Operatoren miteinander verschmilzt. Wegen der erwähnten Überprüfung der Verschmelzbarkeit kommen hierfür natürlich nur bestimmte Graphmuster-Operatoren in Frage. Diese müssen

- über derselben Datenquelle (siehe Definition 3.22 auf Seite 28) operieren,
- nur einen Ausgangs-Variablendatenfluss haben und
- durch denselben Verbund-Operator oder Baum von Verbund-Operatoren miteinander verbunden sein.

Bezüglich des Verbundes der beiden Kandidaten gelten die folgenden zusätzlichen Bedingungen.

- Im Fall eines einzelnen Verbund-Operators dürfen die beiden Eingangs-Variablendatenflüsse, welche jeweils von einem der beiden Graphmuster-Operatoren bereitgestellt werden, nicht optional sein.
- Im Fall eines Verbund-Baumes enthält dieser einen Teilbaum, dessen Wurzel der kleinste gemeinsame Vorfahre der beiden Graphmuster-Operatoren ist. Dieser Teilbaum muss die folgenden Bedingungen erfüllen.
  - Für beide Graphmuster-Operatoren darf keiner der Variablendatenflüsse, die sich auf dem jeweiligen Pfad vom Graphmuster-Operatoren zum Wurzel-Operator des Teilbaums befinden, optional sein.
  - Keiner der Verbund-Operatoren, die auf einem der beiden Pfade von den beiden Graphmuster-Operatoren zum Wurzel-Operator des Teilbaums liegen, darf weitere Ausgangs-Variablendatenflüsse haben.
  - Keiner der Verbund-Operatoren, die auf einem der beiden Pfade von den beiden Graphmuster-Operatoren zum Wurzel-Operator des Teilbaums liegen, darf eine RDF-Graph-Klausel (siehe Abschnitt 2.3.7 ab Seite 12) repräsentieren. Nach der Erzeugungsvorschrift für die einzelnen Graphmuster (siehe Abschnitt 4.1.2 ab Seite 41) repräsentiert ein Verbund-Operator eine RDF-Graph-Klausel, wenn seine Menge an Datenquellen genau einen graph-bereitstellenden Operator enthält und dieser nicht den Default-Graph der RDF-Datenmenge repräsentiert.

Anhand der genannten Bedingungen wurde der in Abbildung 6.1 (Seite 95) dargestellte Algorithmus FindMergeableGPOs entworfen. Er findet in einem SQGM zwei Graphmuster-Operatoren die unter den genannten Bedingungen miteinander verschmelzbar sind, sofern das SQGM solche Operatoren enthält.

Hierzu untersucht er zunächst mögliche Kandidaten, welche über einen einzelnen Verbund-Operator miteinander verbunden sind. Sollten dabei keine verschmelzbaren Graphmuster-Operatoren gefunden werden, so iteriert der Algorithmus über alle Verbund-Bäume des SQGM. Hierbei wird mit Hilfe des rekursiven Algorithmus FindCandidateGPOs (siehe Abbildung 6.2 auf Seite 96) zunächst für den jeweils aktuell betrachteten Verbund-Baum die Menge aller über diesen Baum verbundenen Graphmuster-Operatoren bestimmt, welche für eine Verschmelzung in Frage kommen. Diese Menge wird daraufhin nach zwei tatsächlich verschmelzbaren Operatoren durchsucht. Sollte kein solches Paar vorhanden sein, so wird die Suche mit dem nächsten Verbund-Baum fortgesetzt.

Es sei noch darauf hingewiesen, dass der beschriebene Algorithmus FindMergeableGPOs die folgende Eigenschaft besitzt. Wenn FindMergeableGPOs ein Paar (o_1, o_2) verschmelzbare Graphmuster-Operatoren findet, so wird o_1 bei einer links-assoziativen Tiefensuche im entsprechenden Verbund-Baum immer zuerst gefunden. Demzufolge gilt für den Wurzel-Verbund-Operator des o_1 und o_2 direkt verbindenden Teilbaumes, dass o_1 immer einen Teil seines linken Eingangs-Variablendatenflusses bereitstellt und o_2 einen Teil seines rechten.

```
ALGORITHMUS: FINDMERGEABLEGPOS
EINGABE: q := (OP, DF, r, dflt, NG) – das untersuchte SQGM
AUSGABE: (o_1, o_2) – ein Paar mit zwei verschmelzbaren Graphmuster-Operatoren

// Möglichkeit 1 - über einen Verbund-Operator verbundene Graphmuster-Operatoren
foreach ( jo ∈ JO ⊂ OP ) do
        (i_left, i_right) := jo.input;
        if ( i_left.optional ≠ T und i_right.optional ≠ T ) then
                o_left :=  ProvOp ( i_left);
                o_right :=  ProvOp ( i_right);
                if ( |o_left.output| = 1 und |o_right.output| = 1 ) then
                        if ( o_left ∈ GPO ⊂ OP und o_right ∈ GPO ⊂ OP und DS(o_left) = DS(o_right) ) then
                                return (o_left, o_right);

// Möglichkeit 2 - über einen Verbund-Baum verbundene Graphmuster-Operatoren
Sei R ⊆ JO ⊂ OP die Menge der Wurzeln aller Verbund-Bäume;
foreach ( jo ∈ R ) do

        // finde alle über jo's Baum verbundenen und in Frage kommenden Graphmuster-Operatoren
        M := FINDCANDIDATEGPOS ( jo );

        foreach ( (o_1, o_2) ∈ M × M mit o_1 ≠ o_2 ) do
                if ( DS(o_1) = DS(o_2) ) then
                        return (o_1, o_2);

// keine verschmelzbaren Graphmuster-Operatoren gefunden
return undef.;
```

Abbildung 6.1: Algorithmus zum Finden von zwei verschmelzbaren Graphmuster-Operatoren

Mit den entworfenen Algorithmen ist es möglich die notwendige Restrukturierungsregel für die betrachtete Heuristik zu spezifizieren.

Restrukturierungsregel 6.1 – MERGEALLJOINEDGPOS

Ziel: Diese Regel verfolgt das Ziel alle verschmelzbaren Graphmuster-Operatoren eines SQGM miteinander zu verschmelzen.

Schritte: Zur Erreichung des Ziels müssen schrittweise jeweils zwei verschmelzbare Graphmuster-Operatoren miteinander verschmolzen werden.

Algorithmen: Um festzustellen, ob ein weiterer Restrukturierungsschritt für diese Regel ausführbar ist, kann der Algorithmus FINDMERGEABLEGPOS (siehe Abbildung 6.1) eingesetzt werden. Findet dieser Algorithmus zwei verschmelzbare Graphmuster-Operatoren, so ist ein weiterer Schritt ausführbar und ansonsten nicht.
Zur Erstellung der initialen Sequenz von Transformationsregelinstanzen für die Ausführung eines Schrittes kann der Algorithmus CREATERULESTOMERGEJOINEDGPOS (siehe Abbildung 6.3 auf Seite 96) genutzt werden. Er muss mit dem, durch den Algorithmus FINDMERGEABLEGPOS gefundenen Paar verschmelzbarer Graphmuster-Operatoren gestartet werden. □

Der in der Regeldefinition erwähnte Algorithmus CREATERULESTOMERGEJOINEDGPOS stellt für ein Paar verschmelzbarer Graphmuster-Operatoren, wie es vom FINDMERGEABLEGPOS-Algorithmus (siehe Abbildung 6.1) gefunden wird, eine Sequenz von Regelinstanzen zusammen, deren Abarbeitung zur Verschmelzung von der beiden Operatoren führt. Die entsprechenden Transformationsregeln und die Funktionsweise des Algorithmus werden im folgenden Abschnitt vorgestellt.

ALGORITHMUS: FINDCANDIDATEGPOS
EINGABE: $q := (OP, DF, r, dflt, NG)$ – das untersuchte SQGM
EINGABE: $jo \in JO \subset OP$ – Wurzel des zu untersuchenden Verbund-Baumes
AUSGABE: M – gefundene Kandidaten

$M := \emptyset$;
$(i_{\text{left}}, i_{\text{right}}) := jo.\text{input}$;
$o_{\text{left}} := \mathsf{ProvOp}(i_{\text{left}})$;
$o_{\text{right}} := \mathsf{ProvOp}(i_{\text{right}})$;

// Betrachten des linken Teilbaums
`if` ($i_{\text{left}}.\text{optional} \neq \mathsf{T}$) `then`
    `if` ($o_{\text{left}} \in GPO \subset OP$ und $|o_{\text{left}}.\text{output}| = 1$) `then`
        $M := M \cup \{o_{\text{left}}\}$;
    `if` ($o_{\text{left}} \in JO \subset OP$ und $|o_{\text{left}}.\text{output}| = 1$) `then`
        `if` ($|\mathsf{DS}(o_{\text{left}})| \neq 1$ oder $\mathsf{DS}(o_{\text{left}}) = \{dflt\}$ oder $\mathsf{DS}(o_{\text{left}}) = \mathsf{DS}(jo)$) `then`
            $M := M \cup$ FINDCANDIDATEGPOS (o_{left});

// Betrachten des rechten Teilbaums
`if` ($i_{\text{right}}.\text{optional} \neq \mathsf{T}$) `then`
    `if` ($o_{\text{right}} \in GPO \subset OP$ und $|o_{\text{right}}.\text{output}| = 1$) `then`
        $M := M \cup \{o_{\text{right}}\}$;
    `if` ($o_{\text{right}} \in JO \subset OP$ und $|o_{\text{right}}.\text{output}| = 1$) `then`
        `if` ($|\mathsf{DS}(o_{\text{right}})| \neq 1$ oder $\mathsf{DS}(o_{\text{right}}) = \{dflt\}$ oder $\mathsf{DS}(o_{\text{right}}) = \mathsf{DS}(jo)$) `then`
            $M := M \cup$ FINDCANDIDATEGPOS (o_{right});

`return` M;

Abbildung 6.2: Rekursiver Algorithmus zum Finden aller Graphmuster-Operatoren, die über einen Verbund-Baum verbunden sind und für eine Verschmelzung in Frage kommen

ALGORITHMUS: CREATERULESTOMERGEJOINEDGPOS
EINGABE: $(o_1, o_2) := GPO \times GPO$ – ein Paar mit zwei verschmelzbaren Graphmuster-Operatoren, wobei o_1 bei einer links-assoziativen Tiefensuche zuerst gefunden wird
AUSGABE: $rules$ – Liste von Regelinstanzen

$rules := \emptyset$;
`if` (o_1 und o_2 sind über einen Verbund-Operator jo miteinander verbunden) `then`
    Erzeuge Instanz r einer Regel zum Verschmelzen der Eingangs-Operatoren für jo;
    Hänge r an das Ende von $rules$;

`else`
    Sei $root \in JO$ die Wurzel des o_1 und o_2 direkt verbindenen Verbund-Baumes;
    Sei $path_1$ eine Liste mit allen Verbund-Operatoren von $root$ bis zum direkt über o_1 befindlichen Operator;
    Sei $path_2$ eine Liste mit allen Verbund-Operatoren von $root$ bis zum direkt über o_2 befindlichen Operator;

    // Schritt 0
    Stelle sicher, dass o_1 den linken Eingang für den letzten Verbund-Operator in $path_1$ bereitstellt;
    Stelle sicher, dass o_2 den rechten Eingang für den letzten Verbund-Operator in $path_2$ bereitstellt;

    // Schritt 1
    `if` ($|path_2| > 1$) `then` // wenn o_2 nicht direkt den rechten Eingang von $root$ bereitstellt
        Erweitere $rules$ um Regelinstanzen, durch welche der Verbund-Baum restrukturiert wird, so dass o_2 direkt den rechten Eingang von $root$ bereitstellt;

    // Schritt 2
    `if` ($|path_1| > 1$) `then` // wenn o_1 nicht direkt den linken Eingang von $root$ bereitstellt
        Erweitere $rules$ um Regelinstanzen, durch welche der Verbund-Baum restrukturiert wird, so dass o_2 direkt den rechten Eingang für den letzten Verbund-Operator in $path_1$ bereitstellt;

    // Schritt 3
    Erweitere $rules$ um die Instanz einer Regel zum Verschmelzen der Eingangs-Operatoren des letzten Verbund-Operators in $path_1$;

Abbildung 6.3: Algorithmus zum Erzeugen einer Sequenz von Regelinstanzen, um zwei verbundene Graphmuster-Operatoren zu verschmelzen

6.3.3 Notwendige Transformationsregeln

Anhand der Beschreibung der Funktionsweise des CreateRulesToMergeJoinedGPOs-Algorithmus (siehe Abbildung 6.3 auf Seite 96) werden in diesem Abschnitt die hierfür notwendigen Transformationsregeln definiert. Zur Erzeugung einer Sequenz von Regelinstanzen für die Verschmelzung eines Paars (o_1, o_2) verbundener Graphmuster-Operatoren unterscheidet der Algorithmus zunächst, ob o_1 und o_2 direkt durch einen Verbund-Operator verbunden sind oder durch einen Baum von Verbund-Operatoren.

Sind die beiden zu verschmelzenden Operatoren direkt durch einen Verbund-Operator verbunden, so können o_1 und o_2 zu einem Graphmuster-Operator verschmolzen werden, der im SQGM den Verbund-Operator ersetzt. Für diese Verschmelzung wurde die im Folgenden definierte Transformationsregel MergeJoinedGPOs entwickelt. Der CreateRulesToMergeJoinedGPOs-Algorithmus muss der Sequenz von Regelinstanzen lediglich eine, den Verbund-Operator betreffende, MergeJoinedGPOs-Instanz hinzufügen.

Transformationsregel 6.1 – MergeJoinedGPOs

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $o_{\text{left}} \in GPO$
ii) $o_{\text{right}} \in GPO$
iii) $i_{\text{left}}.\text{optional} \neq \mathsf{T}$
iv) $i_{\text{right}}.\text{optional} \neq \mathsf{T}$
v) $|o_{\text{left}}.\text{output}| = 1$
vi) $|o_{\text{right}}.\text{output}| = 1$
vii) $\mathsf{DS}(o_{\text{left}}) = \mathsf{DS}(o_{\text{right}})$

Anwendung: Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt.

i) $o_{\text{left}}.\texttt{triplePatterns} := o_{\text{left}}.\texttt{triplePatterns} \circ o_{\text{right}}.\texttt{triplePatterns}$
ii) $o_{\text{left}}.\texttt{constraints} := o_{\text{left}}.\texttt{constraints} \circ o_{\text{right}}.\texttt{constraints}$
iii) $o_{\text{left}}.\texttt{contr} := o_{\text{left}}.\texttt{contr} \vee o_{\text{right}}.\texttt{contr}$
iv) $o_{\text{left}}.\text{output} := o.\text{output}$
v) $o.\text{output} := ()$
vi) $DF' := DF \setminus \{i_{\text{left}}, i_{\text{right}}, o_{\text{right}}.\text{input}\}$
vii) $OP' := OP \setminus \{o, o_{\text{right}}\}$

Erläuterung: Diese Regel verschmilzt die Operatoren o, o_{left} und o_{right} miteinander wenn die folgenden Bedinungen erfüllt sind. Die Eingangsdatenflüsse i_{left} und i_{right} des betroffenen Verbund-Operators o dürfen nicht optional sein. Die beiden bereitstellenden Operatoren o_{left} und o_{right} müssen Graphmuster-Operatoren ohne weitere Ausgangsdatenflüsse sein und sie müssen über derselben Datenquelle operieren. Für die Verschmelzung werden die Eigenschaften von o_{left} um die von o_{right} ergänzt, die Verbindungen zwischen den drei Operatoren entfernt, o durch o_{left} ersetzt und o und o_{right} aus dem SQGM entfernt.

Begründung: Der Operator o repräsentiert den Verbund von zwei gruppierten Graphmustern. Diese beiden Graphmuster sind jeweils nur einfache Graphmuster mit Wertebeschränkungen und befinden sich im selben Gültigkeitsbereich (siehe Abschnitt 2.3.7 ab Seite 12), wie die folgenden Ausführungen zeigen.

Nach den Voraussetzungen i) und ii) können o_{left} und o_{right} jeweils nur noch 1.) einfache Graphmuster und Wertebeschränkungen, 2.) optionale Graphmuster oder 3.) RDF-Graph-Klauseln sein. Der zweite Fall wird durch die Voraussetzungen iii) und iv) ausgeschlossen.

Der dritte Fall kann vernachlässigt werden, da sich beide Graphmuster wegen der Voraussetzung vii) im Gültigkeitsbereich derselben RDF-Graph-Klausel befinden. Die Voraussetzungen v) und vi) garantieren weiterhin, dass keines der beiden Graphmuster in einem weiteren Zusammenhang verwendet wird.
Die beiden durch q und q' repräsentierten Anfragen sind semantisch äquivalent, weil die Operatoren o, o_{left} und o_{right} verschmolzen werden können, ohne das sich die Lösungsmenge ändert. Nach der SPARQL-Spezifikation ist s eine Lösung einer Graphmuster-Gruppe, wenn sie Lösung aller gruppierten Graphmuster ist. Sei nun s eine Lösung der durch q repräsentierten Anfrage. Somit ist s auch Lösung aller Tripelmuster und Wertebeschränkungen in o_{left} und o_{right} vor der Transformation. Entsprechend der Anwendungsvorschrift ist s auch Lösung aller Tripelmuster und Wertebeschränkungen in o_{left} nach der Transformation und damit auch von der durch q' repräsentierten Anfrage. Sei s andererseits eine Lösung der durch q' repräsentierten Anfrage. Dann ist s auch eine Lösung aller Tripelmuster und Wertebeschränkungen in o_{left} nach der Transformation. Entsprechend der Anwendungsvorschrift ist s auch Lösung aller Tripelmuster und Wertebeschränkungen in o_{left} und o_{right} vor der Transformation. Somit ist s auch eine Lösung der von q repräsentierten Anfrage.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Beispiel 6.2 *In der Abbildung 6.4.a ist ein SQGM dargestellt. Wenn auf den darin enthaltenen Verbund-Operator die Transformationsregel* MergeJoinedGPOs *angewandt wird, erhält man das in Abbildung 6.4.b dargestellte SQGM.*

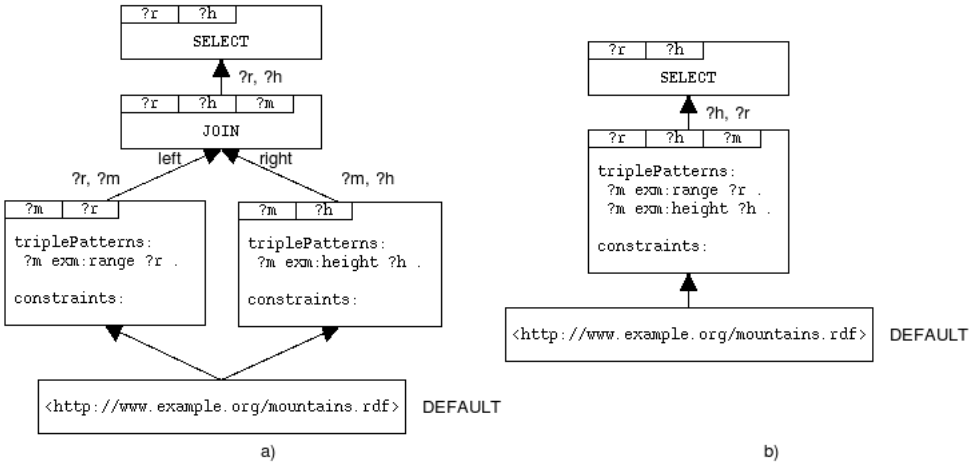


Abbildung 6.4: Anwendung der Transformationsregel MergeJoinedGPOs auf ein SQGM □

Bei der Ausführung des Algorithmus CreateRulesToMergeJoinedGPOs besteht die zweite Möglichkeit wie bereits erwähnt darin, dass die zu verschmelzenden Graphmuster-Operatoren o_1 und o_2 durch einen Baum von Verbund-Operatoren verbunden sind. In diesem Fall wird das Verschmelzen etwas aufwendiger, da die beiden Operatoren zunächst durch eine Restrukturierung des Verbund-Baumes zusammengebracht werden müssen, um wieder die Regel MergeJoinedGPOs anwenden zu können. Im Folgenden werden zunächst alle für die Rotation des Verbund-Baumes notwendigen Transformationsregeln definiert und danach die Restrukturierungsstrategie unter Benutzung dieser Regeln vorgestellt.

Transformationsregel 6.2 – SwitchJoinInputs

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Es gelte:

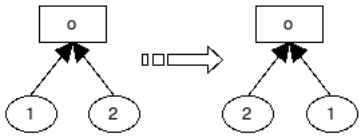
i) $o.\text{input} \neq \emptyset$

Anwendung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$.

Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{input} := (i_{\text{right}}, i_{\text{left}})$

Erläuterung: Diese Regel vertauscht die Eingangsdatenflüsse i_{left} und i_{right} des Verbund-Operators o wie die folgende Abbildung andeutet.



Begründung: Der Operator o repräsentiert den Verbund von zwei gruppierten Graphmustern. Ein Vertauschen der Eingangsdatenflüsse entspricht dem Vertauschen der Position dieser beiden gruppierten Graphmustern innerhalb der Gruppe. Nach der SPARQL-Spezifikation ist s eine Lösung einer Graphmuster-Gruppe, wenn sie Lösung aller gruppierten Graphmuster ist. Eine Änderung der Reihenfolge dieser gruppierten Graphmuster ändert daran nichts. Deshalb sind die durch q und q' repräsentierten Anfragen semantisch äquivalent.

Ergänzung: Es sei darauf hingewiesen, dass das Vertauschen der Eingänge eines Verbund-Operators mit der Äquivalenzbeziehung

$$(A_1 \bowtie A_2) = (A_2 \bowtie A_1)$$

der relationalen Algebra vergleichbar ist. □

Transformationsregel 6.3 – SwitchJoinedJoinRightInputs

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$ und $o_{\text{left}} := ProvOp(i_{\text{left}})$. Es gelte:

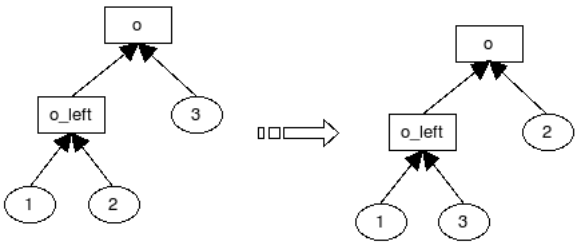
i) $i_{\text{left}}.\text{optional} \neq \mathsf{T}$
ii) $o_{\text{left}} \in JO$
iii) $o_{\text{left}}.\text{input} \neq ()$
iv) $|o_{\text{left}}.\text{output}| = 1$

Anwendung: Seien $(j_{\text{left}}, j_{\text{right}}) := o_{\text{left}}.\text{input}$.

Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o_{\text{left}}.\text{input} := (j_{\text{left}}, i_{\text{right}})$
ii) $o_{\text{left}}.\text{provVars} := j_{\text{left}}.\text{vars} \cup i_{\text{right}}.\text{vars}$
iii) $i_{\text{left}}.\text{vars} := j_{\text{left}}.\text{vars} \cup i_{\text{right}}.\text{vars}$
iv) $o.\text{input} := (i_{\text{left}}, j_{\text{right}})$

Erläuterung: Wenn für einen Verbund-Operator o der linke Eingangsdatenfluss i_{left} nicht optional ist und von einem zusätzlichen Verbund-Operator o_{left} ohne weitere Ausgangsdatenflüsse bereitgestellt wird, dann vertauscht diese Regel die beiden rechten Eingangsdatenflüsse i_{right} und j_{right} der beiden Operatoren wie die folgende Abbildung andeutet.



Begründung: Der Operator o repräsentiert den Verbund von zwei gruppierten Graphmustern. Eines davon wird durch o_{left} repräsentiert und entspricht einer Gruppe von zwei weiteren Graphmustern. Durch die Voraussetzung i) wird ausgeschlossen, dass diese zweite Gruppe ein optionales Graphmuster ist. Darum kann die Einheit aus den beiden Verbund-Operatoren o und o_{left} auch als Gruppe von drei Graphmustern angesehen werden.
Ein Vertauschen der beiden rechten Eingangsdatenflüsse entspricht dem Vertauschen der Position der beiden, durch die entsprechend bereitstellenden Operatoren repräsentierten, Graphmuster innerhalb der Gruppe. Nach der SPARQL-Spezifikation ist s eine Lösung einer Graphmuster-Gruppe, wenn sie Lösung aller gruppierten Graphmuster ist. Eine Änderung der Reihenfolge dieser gruppierten Graphmuster ändert daran nichts. Deshalb sind die durch q und q' repräsentierten Anfragen semantisch äquivalent.

Ergänzung: Es sei darauf hingewiesen, dass das Vertauschen der beiden rechten Eingänge von verbundenen Verbund-Operatoren eine Änderung der Ausführungsreihenfolge bewirkt und ein Analogon zu der Äquivalenzbeziehung

$$\big((A_1 \bowtie A_2) \bowtie A_3)\big) = \big((A_1 \bowtie A_3) \bowtie A_2\big)$$

der relationalen Algebra darstellt.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Transformationsregel 6.4 – RerootJoinTreeRight

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

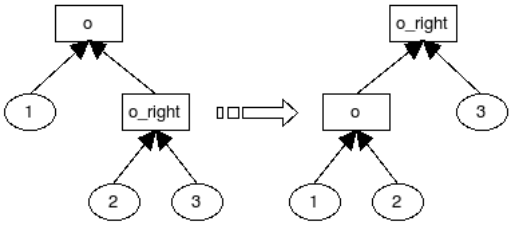
i) $i_{\text{right}}.\text{optional} \neq \mathsf{T}$
ii) $o_{\text{right}} \in JO$
iii) $o_{\text{right}}.\text{input} \neq ()$
iv) $|o_{\text{right}}.\text{output}| = 1$

Anwendung: Seien $(j_{\text{left}}, j_{\text{right}}) := o_{\text{right}}.\text{input}$.

Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP, DF, r, dflt, NG)$ wie folgt.

i) $o.\text{input} := (i_{\text{left}}, j_{\text{left}})$
ii) $o.\text{provVars} := i_{\text{left}}.\text{vars} \cup j_{\text{left}}.\text{vars}$
iii) $i_{\text{right}}.\text{vars} := i_{\text{left}}.\text{vars} \cup j_{\text{left}}.\text{vars}$
iv) $o_{\text{right}}.\text{output} := o.\text{output}$
v) $o.\text{output} := (i_{\text{right}})$
vi) $o_{\text{right}}.\text{input} := (i_{\text{right}}, j_{\text{right}})$

Erläuterung: Wenn für einen Verbund-Operator o der rechte Eingangsdatenfluss i_{right} nicht optional ist und von einem zusätzlichen Verbund-Operator o_{right} ohne weitere Ausgangsdatenflüsse bereitgestellt wird, dann rotiert diese Regel die beiden Operatoren so, dass o einen Eingangsdatenfluss für o_{right} bereitstellt, wie die folgende Abbildung andeutet. Die Richtung des Variablendatenflusses i_{right}, welcher o und o_{right} verbindet, wird dabei umgekehrt.



Begründung: Der Operator o repräsentiert den Verbund von zwei gruppierten Graphmustern. Eines davon wird durch o_{right} repräsentiert und entspricht einer Gruppe von zwei weiteren Graphmustern. Durch die Voraussetzung i) wird ausgeschlossen, dass diese zweite Gruppe ein optionales Graphmuster ist. Darum kann die Einheit aus den beiden Verbund-Operatoren o und o_{left} auch als Gruppe von drei Graphmustern angesehen werden. Da das Rotieren der beiden Verbund-Operatoren nichts an dieser Gruppe ändert, sind die durch q und q' repräsentierten Anfragen semantisch äquivalent.

Ergänzung: Es sei an dieser Stelle darauf hingewiesen, dass das Rotieren der beiden Verbund-Operatoren eine Änderung der Ausführungsreihenfolge bewirkt und ein Analogon zu der Äquivalenzbeziehung

$$\big(A_1 \bowtie (A_2 \bowtie A_3)\big) = \big((A_1 \bowtie A_2) \bowtie A_3\big)$$

der relationalen Algebra darstellt.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Mit Hilfe der oben definierten Transformationsregeln ist es jetzt möglich den noch ausstehenden Teil des Algorithmus CreateRulesToMergeJoinedGPOs zu beschreiben. Wie bereits erwähnt besteht die Teilaufgabe darin die beiden zu verschmelzenden Graphmuster-Operatoren o_1 und o_2, welche über einen Verbundbaum verbunden sind, zusammenzuführen. Diese Zusammenführung beinhaltet eine Restrukturierung des Verbund-Baumes, bis o_1 und o_2 über denselben Verbund-Operator verbunden sind.

Die Abbildung 6.5.a deutet beispielhaft die Ausgangslage an, wobei die grau hinterlegten, rechteckigen Symbole Verbund-Operatoren repräsentieren und die beiden anderen rechteckigen Symbole repräsentieren die beiden zu verschmelzenden Graphmuster-Operatoren. Durch die Ovale werden weitere verbundene Operatoren symbolisiert, deren Typ für die folgenden Betrachtungen unwichtig ist. Es kann sich hierbei auch um weitere Teile des Verbund-Baumes handeln.

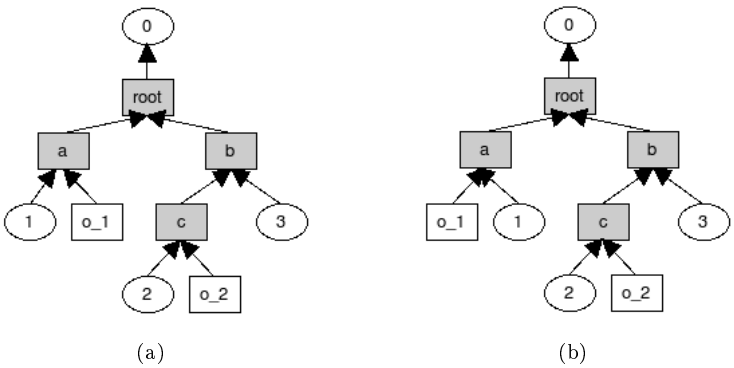


Abbildung 6.5: Beispielhaft angedeuteter Verbund-Baum bei der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Beginn)

Der beispielhaft dargestellte Baum zeigt lediglich den Verbund-Baum, welcher o_1 und o_2 direkt miteinander verbindet. Er kann Teil eines größeren Verbund-Baumes sein, in dem *root* der kleinste gemeinsame Vorfahre von o_1 und o_2 ist. Für den Algorithmus muss jedoch nur dieser, direkt verbindende Teilbaum restrukturiert werden.

In dem hier beschriebenen Teil des Algorithmus werden zunächst die beiden Listen $path_1$ und $path_2$ initialisiert. Diese beinhalten jeweils alle Verbund-Operatoren des direkt verbindenen Teilbaums von *root* bis zum direkt über o_1 bzw. o_2 befindlichen Operator. Für das angedeutete Beispiel sind das $(root, a)$ und $(root, b, c)$. Daraufhin wird als Voraussetzung für die weiteren Schritte sicher gestellt, dass o_1 den linken Eingang für den letzten Verbund-Operator in $path_1$ bereitstellt und o_2 den rechten Eingang für den letzten Verbund-Operator in $path_2$. Diese Sicherstellung ist in der Pseudocode-Darstellung des Algorithmus CreateRulesToMergeJoinedGPOs (siehe Abbildung 6.3 auf Seite 96) als nullter Schritt gekennzeichnet. Hierfür können wenn nötig Instanzen der Regel SwitchJoinInputs (siehe Definition 6.2 auf Seite 98) erzeugt werden. Die Abbildung 6.5.b deutet den Beispiel-Baum nach deren Abarbeitung an.

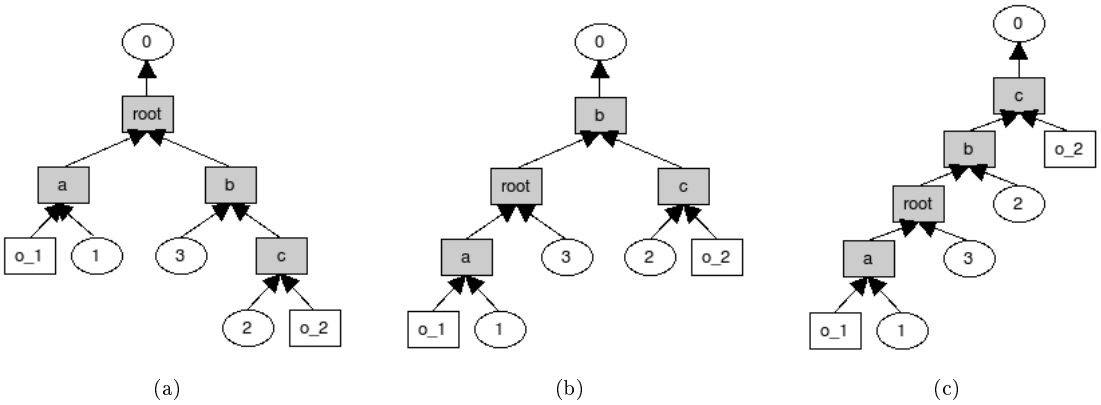


Abbildung 6.6: Beispielhaft angedeuteter Verbund-Baum während der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Fortsetzung)

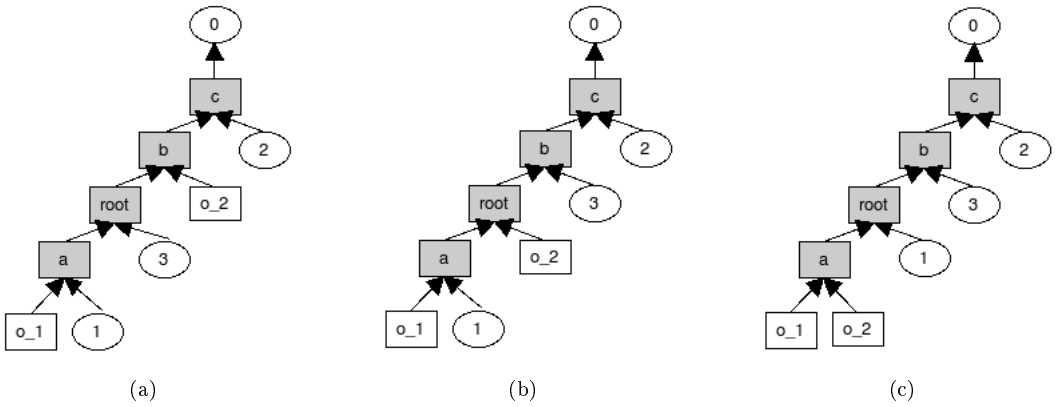


Abbildung 6.7: Beispielhaft angedeuteter Verbund-Baum während der Abarbeitung des Algorithmus CreateRulesToMergeJoinedGPOs (Abschluss)

Nach der Vorbereitung wird im ersten Schritt des Algorithmus dafür gesorgt, dass o_2 direkt den rechten Eingang-Variablendatenfluss des Wurzel-Verbund-Operators *root* bereitstellt. Hierfür wird der Verbund-Baum zunächst so rotiert, dass der direkt über o_2 befindliche Verbund-Operator letztendlich zur neuen Wurzel wird, wie die Abbildungen 6.6.a, 6.6.b und 6.6.c andeuten. Dafür

müssen entsprechende Instanzen der Regeln SwitchJoinInputs (siehe Definition 6.2 auf Seite 98) und RerootJoinTreeRight (siehe Definition 6.4 auf Seite 100) erzeugt werden. Danach wird o_2 bis zum Operator *root* im Verbundbaum nach unten propagiert, wie die Abbildungen 6.7.a und 6.7.b andeuten. Dazu werden Instanzen der Regel SwitchJoinedJoinRightInputs (siehe Definition 6.3 auf Seite 99) benötigt.

Im zweiten Schritt des Algorithmus wird dafür gesorgt, dass o_2 direkt den rechten Eingang-Variablendatenfluss des direkt über o_1 befindliche Verbund-Operators bereitstellt. Dazu müssen entsprechende SwitchJoinInputs- und SwitchJoinedJoinRightInputs-Instanzen erzeugt werden. Deren Abarbeitung deutet die Abbildung 6.7.c an.

Nach dem zweiten Schritt ist das Teilziel des Algorithmus erfüllt und es kann wie angedeutet eine MergeJoinedGPOs-Instanz erzeugt werden, um o_1 und o_2 endgültig zu verschmelzen.

Für den Algorithmus CreateRulesToMergeJoinedGPOs (siehe Abbildung 6.3 auf Seite 96) wurde eine verfeinerte Version entwickelt. Diese wurde hauptsächlich um die Erwähnung der eingesetzten Transformationsregelinstanzen erweitert und ist in Abbildung 6.8 (Seite 104) dargestellt.

Zum Abschluss dieses Abschnitts sind in der Tabelle 6.1 die möglichen Folgeregeln der definierten Transformationsregeln aufgelistet.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verw. Operator
MergeJoinedGPOs	–	FindContradiction	o_{left}
	–	RemoveEqualTriplePatterns	o_{left}
SwitchJoinedJoinRightInputs	–	PropagateUnreqVarJoin	o
RerootJoinTreeRight	–	PropagateUnreqVarJoin	o

Tabelle 6.1: Mögliche Folgeregeln der Transformationsregeln des Abschnitts 6.3.3

6.3.4 Spezifikation der Heuristik

Mit Hilfe der in den vorherigen Abschnitten spezifizierten Regeln kann die, auf den in Abschnitt 6.3.1 (ab Seite 92) dargelegten Vermutungen basierende Heuristik spezifiziert werden.

Heuristik 6.1 Es wird angenommen, dass für die meisten SPARQL-Anfragen, welche durch ein den Voraussetzungen entsprechendes SQGM repräsentiert werden können, die Ausführung unter den vorausgesetzen Bedingungen schneller durchgeführt werden kann, wenn das SQGM mit Hilfe der angegebenen Restrukturierungsregel restrukturiert wird.
Effizienzkriterium: Ausführungszeit
Restrukturierungsregeln: MergeAllJoinedGPOs
Voraussetzungen:

- Die Anfrage wird mit der Standard-Anfragebearbeitungskomponente von ARQ ausgeführt.
- Die Anfrage wird direkt über einer persistent gespeicherten RDF-Datenmenge ausgeführt.
- Das SQGM enthält mehrere über Verbund-Operatoren oder Bäume von Verbund-Operatoren miteinander verbundene Graphmuster-Operatoren.
- Alle Graphmuster-Operatoren des SQGM enthalten lediglich Tripelmuster.
- Die in allen Graphmuster-Operatoren des SQGM enthalten Tripelmuster können während der Ausführung des Fast-Path-Algorithmus in Lösungsstufen zusammengefasst werden. □

ALGORITHMUS: CREATERULESTOMERGEJOINEDGPOS
EINGABE: $(o_1, o_2) := GPO \times GPO$ – ein Paar mit zwei verschmelzbaren Graphmuster-Operatoren, wobei o_1 bei einer links-assoziativen Tiefensuche zuerst gefunden wird
AUSGABE: $rules$ – Liste von Regelinstanzen

```
rules := ∅;
if ( o1 und o2 sind über einen Verbund-Operator jo miteinander verbunden ) then
        Erzeuge eine MergeJoinedGPOs-Instanz für jo und hänge diese an das Ende von rules;
else
        Sei root ∈ JO die Wurzel des o1 und o2 direkt verbindenen Verbund-Baumes;
        Sei path1 eine Liste mit allen Verbund-Operatoren von root bis zum direkt über o1 befindlichen Operator;
        Sei path2 eine Liste mit allen Verbund-Operatoren von root bis zum direkt über o2 befindlichen Operator;

        // Schritt 0
        Sei i der linke Eingangs-Variablendatenfluss des letzten Verbund-Operators in path1;
        if ( ProvOp(i) ≠ o1 ) then
                Erzeuge eine SwitchJoinInputs-Instanz für ProvOp(i) und hänge diese an das Ende von rules;

        Sei i der rechte Eingangs-Variablendatenfluss des letzten Verbund-Operators in path2;
        if ( ProvOp(i) ≠ o2 ) then
                Erzeuge eine SwitchJoinInputs-Instanz für ProvOp(i) und hänge diese an das Ende von rules;

        // Schritt 1
        if ( |path2| > 1 ) then          // wenn o2 nicht direkt den rechten Eingang von root bereitstellt
                for ( i := 1, ··· , |path2| − 1 ) do
                        Sei jo_i der i-te und jo_{i+1} der (i + 1)-te Verbund-Operator in path2;
                        Sei d der rechte Eingangs-Variablendatenfluss von jo_i ;
                        if ( ProvOp(d) ≠ jo_{i+1} ) then
                                Erzeuge eine SwitchJoinInputs-Instanz für jo_i und hänge diese an rules;
                        Erzeuge eine RerootJoinTreeRight-Instanz für jo_i und hänge diese an rules;

                for ( i := |path2| , ··· , 2 ) do
                        Erzeuge eine SwitchJoinedJoinRightInputs-Instanz für den i-ten Verbund-Operator in
                        path2 und hänge sie an das Ende von rules;

        // Schritt 2
        if ( |path1| > 1 ) then          // wenn o1 nicht direkt den linken Eingang von root bereitstellt
                for ( i := 1, ··· , |path1| − 2 ) do
                        Sei jo_i der i-te, jo_{i+1} der i + 1-te und jo_{i+2} der (i + 2)-te Verbund-Operator in path1;
                        Sei d der linke Eingangs-Variablendatenfluss von jo_{i+1} ;
                        if ( ProvOp(d) ≠ jo_{i+2} ) then
                                Erzeuge eine SwitchJoinInputs-Instanz für jo_{i+1} und hänge diese an rules;
                        Erzeuge eine SwitchJoinedJoinRightInputs-Instanz für jo_i und hänge diese an rules;
                Erzeuge eine SwitchJoinedJoinRightInputs-Instanz für den vorletzten Verbund-Operator in
                path1 und hänge diese an rules;

        // Schritt 3
        Erzeuge eine MergeJoinedGPOs-Instanz für den letzten Verbund-Operator in path1 und hänge diese an
        das Ende von rules;
```

Abbildung 6.8: Verfeinerter Algorithmus zum Erzeugen einer Sequenz von Regelinstanzen, um zwei verbundene Graphmuster-Operatoren zu verschmelzen

6.3.5 Experimentelle Untersuchung

Um die Heuristik 6.1 untersuchen zu können, wurden zunächst die in den Abschnitten 6.3.2 und 6.3.3 spezifizierten Regeln im Rahmen der in Kapitel 5 vorgestellten Umgebung implementiert. Danach wurden die Anfragen A.1 bis A.9 (siehe Anhang A ab Seite 124) formuliert. Hierbei wurde besonderer Wert darauf gelegt, dass die Testanfragen ein möglichst breites Spektrum der, im Rahmen der für die Heuristik möglichen Anfragen wiederspiegeln. Alle formulierten Anfragen haben die notwendige Eigenschaft, dass sich die enthaltenen Tripelmuster vom Fast-Path-Algorithmus in eine gemeinsame Gruppe einteilen lassen.

Die Anfragen A.1 bis A.4 sind am simpelsten. Sie bestehen aus einer Graphmuster-Gruppe von zwei einfachen Graphmustern, welche jeweils ein Tripelmuster enthalten. Der Unterschied zwischen diesen Anfragen besteht hauptsächlich in der Art der Beziehung zwischen den abgefragten Ressourcen. Beispielsweise existiert zwischen den `ub:GraduateStudent`- und `ub:GraduateCourse`-Ressourcen eine 1-zu-n-Beziehung, denn jede Ressource vom Typ `ub:GraduateStudent` hat eine oder mehrere `ub:takesCourse`-Eigenschaften (Anfrage A.2). Im Gegensatz dazu hat jede Ressource vom Typ `ub:GraduateStudent` genau eine Eigenschaft `ub:name` (Anfrage A.1). Dieser Unterschied könnte bei der Zeit für die Anfragebearbeitung ein Rolle spielen.

Die Anfragen A.5 und A.6 enthalten jeweils drei oder fünf Tripelmuster, welche auf drei (Anfrage A.5) oder fünf (Anfrage A.6) einfache Graphmuster verteilt sind.

Die restlichen Anfragen sind komplexer als die bereits vorgestellten. Sie enthalten optionale Graphmuster. Die dafür modellierten Operatoren werden entsprechend der in Abschnitt 6.3.2 (ab Seite 93) formulierten Bedingungen bei der Bestimmung von verschmelzbaren Graphmuster-Operatoren durch die Algorithmen FindMergeableGPOs (siehe Abbildung 6.1 auf Seite 95) und FindCandidateGPOs (siehe Abbildung 6.2 auf Seite 96) nicht in Betracht gezogen. Die Anfragen A.7 und A.8 unterscheiden sich hauptsächlich in der speziellen Position der einzelnen Tripelmuster und sind sind semantisch nicht äquivalent. Da jedoch jede `ub:GraduateCourse`-Ressource genau eine `ub:name`-Eigenschaft und mindestens eine `ub:takesCourse`-Eigenschaft hat, haben beide Anfragen über der abgefragten Datenmenge dieselben Ergebnisse. Deshalb eignen sich diese beiden Anfragen sehr gut, um die Heuristik innerhalb komplexerer Anfragen zu untersuchen.

Die komplexeste von den eingesetzten Anfragen ist A.9. Sie enthält eine Graphmuster-Gruppe bestehend aus zwei weiteren Graphmuster-Gruppen und einem einfachen Graphmuster. Die beiden geschachtelten Graphmuster-Gruppen enthalten wiederum jeweils ein einfaches und ein optionales Graphmuster.

Da die Heuristik 6.1 auf die schnellere Bestimmung der Musterübereinstimmungen für die Graphmuster in SPARQL-Anfragen abzielt, wurden für die Tests nur SELECT-Anfragen ohne Klauseln zur Anpassung der Lösungsmenge formuliert. Die Konstruktion eines RDF-Graphen als Ergebnis oder die Anpassung der Lösungsmenge, beispielsweise durch eine DISTINCT- oder ORDERBY-Klausel, würde die Messwerte unnötig beeinflussen.

Für die Testanfragen wurden jeweils zwei Testfälle erstellt und diese mittels eines entsprechenden RDF-Dokuments (siehe Abschnitt 5.6 an Seite 88) beschrieben. Für den jeweils ersten Testfall zu jeder Anfrage wird die eingesetzte Anfragebearbeitungskomponente angewiesen die Normalisierung, Vereinfachung und Restrukturierung des entsprechenden SQGM zu überspringen. Im Gegensatz dazu wird das SQGM bei jedem jeweils zweiten Testfall normalisiert, vereinfacht und unter Anwendung der Restrukturierungsregel MergeAllJoinedGPOs restrukturiert. Auf diese Weise lässt sich die mögliche Effizienzsteigerung bei Anwendung der Heuristik untersuchen.

Die Testfälle wurden mit Hilfe der in Abschnitt 5.6 vorgestellten Testumgebung auf dem in Abschnitt 6.2 (Seite 91) beschriebenen Testsystem für die drei Datenmengen UnivBench(1,0), UnivBench(5,0) und UnivBench(10,0) ausgeführt. Die dabei ermittelten Messwerte für die jeweilige Initialisierungzeit t_{init_u}, Planerzeugungszeit t_{create_u} und Anfrageausführungszeit t_{exec_u} sind in der Tabelle 6.2 (Seite 106) aufgelistet. Die Werte in den $t^*_{x_u}$-Spalten entsprechen den gemessenen Zeiten bei der Anfrageausführung ohne Normalisierung, Vereinfachung und Restrukturierung über der Datenmenge UnivBench(u,0). Die Werte in den t_{x_u}-Spalten entsprechen den gemessenen Zeiten bei der Anfrageausführung mit Normalisierung, Vereinfachung und Restrukturierung über der Datenmenge UnivBench(u,0). Für die jeweiligen Datenmengen UnivBench(u,0) wurde für alle Anfragen das Verhältnis der Anfrageausführungszeiten $t^*_{\text{exec}_u}$ und t_{exec_u} bestimmt und in den r_u-Spalten aufgelistet.

Bei der Untersuchung der ermittelten Messwerte kann zunächst festgestellt werden, dass in allen Fällen die Ausführungsvorbereitungszeit $t_{\text{init}_u} + t_{\text{create}_u}$ im Vergleich zur Anfrageausführungszeit t_{exec_u} vernachlässigbar gering ist und sich die Initialisierungzeiten $t^*_{\text{init}_u}$ und t_{init_u} immer in der selben Größenordnung. Daraus kann gefolgert werden, dass die Ausführung der in dieser Arbeit vorgestellten Algorithmen nur einen sehr geringen Anteil an der gesamten Bearbeitungszeit für eine Anfrage hat.

Anfrage	Messung	$t^*_{x_1}$	t_{x_1}	r_1	$t^*_{x_5}$	t_{x_5}	r_5	$t^*_{x_{10}}$	$t_{x_{10}}$	r_{10}
A.1 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		1 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	2 087 ms	271 ms	ca. 7,7	21 691 ms	9 347 ms	ca. 2,3	51 769 ms	22 134 ms	ca. 2,3
A.2 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	2 152 ms	339 ms	ca. 6,3	25 508 ms	11 332 ms	ca. 2,3	52 555 ms	22 960 ms	ca. 2,3
A.3 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		5 ms	< 0 ms		31 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	23 035 ms	1 905 ms	ca. 12,1	168 851 ms	30 688 ms	ca. 5,5	337 702 ms	63 066 ms	ca. 5,4
A.4 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		3 ms	< 0 ms		5 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	18 949 ms	1 101 ms	ca. 17,2	160 732 ms	11 364 ms	ca. 14,1	348 652 ms	23 758 ms	ca. 14,7
A.5 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	5 684 ms	379 ms	ca. 15,0	38 573 ms	2 553 ms	ca. 15,1	88 070 ms	4 870 ms	ca. 18,1
A.6 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	1 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		11 ms	< 0 ms		9 ms	3 ms	
	Anfrageausführungszeit t_{exec_u}	30 539 ms	820 ms	ca. 37,2	238 451 ms	27 522 ms	ca. 8,7	507 705 ms	59 927 ms	ca. 8,5
A.7 (Seite 124)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		1 ms	1 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		9 ms	< 0 ms		8 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	4 023 ms	4 394 ms	ca. 0,9	27 531 ms	28 947 ms	ca. 1,0	65 597 ms	55 876 ms	ca. 1,2
A.8 (Seite 125)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	5 797 ms	2 517 ms	ca. 2,3	39 426 ms	16 447 ms	ca. 2,4	77 884 ms	32 300 ms	ca. 2,4
A.9 (Seite 125)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	1 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	1 ms	< 0 ms		5 ms	2 ms		4 ms	4 ms	
	Anfrageausführungszeit t_{exec_u}	32 439 ms	9 137 ms	ca. 3,6	254 609 ms	68 048 ms	ca. 3,7	519 513 ms	179 668 ms	ca. 2,9

Legende:
u – repräsentiert durch die Werte 1, 5 bzw. 10 die jeweilige Datenmenge UnivBench(1,0), UnivBench(5,0) bzw. UnivBench(10,0)
x – repräsentiert die Art der gemessenen Zeit, möglich sind 'init', 'create' und 'exec'
$t^*_{x_u}$ – repräsentiert die jeweils für die Datenmenge UnivBench(u,0) gemessene Zeit der Art x bei Bearbeitung ohne Anwendung der Restrukturierungsregeln
t_{x_u} – repräsentiert die jeweils für die Datenmenge UnivBench(u,0) gemessene Zeit der Art x bei Bearbeitung mit Anwendung der Restrukturierungsregeln
r_u – diese Spalten enthalten die jeweiligen Verhältnisse $t^*_{x_u}/t_{x_u}$ der Anfrageausführungszeiten
? – bei der Ermittlung dieser Werte wurde die Messung nach ca. 8 Stunden abgebrochen

Tabelle 6.2: Messwerte der Tests für die Heuristik 6.1

Der Vergleich der jeweiligen $t^*_{\text{exec}_u}$ und t_{exec_u} Werte zeigt, dass bis auf die Ausnahmen bei Anfrage A.7 die Ausführungszeit aller Anfragen durch der Restrukturierung verkürzt wurde. Im Extremfall wurde etwas weniger als 1/37 der Zeit benötigt. Das sind ungefähr 2,7%.

Bei den Anfragen A.1 und A.2 sind sich sowohl die Ausführungszeiten, als auch deren Verbesserungen ähnlich. Das liegt daran, dass im unoptimierten Fall ohne Restrukturierung jeweils genau eine SQL-Anfrage für das erste Tripelmuster ausgeführt wird und danach für alle gefundenen Übereinstimmungen der Variable `?s` jeweils eine weitere SQL-Anfrage. Bei 1 874 `ub:GraduateStudent`-Ressourcen in UnivBench(1,0) sind das also insgesamt 1 875 Datenbankanfragen. Denen steht im optimierten Fall genau eine Anfrage gegenüber, da hier die Tripelmuster in einer Lösungsstufe zusammengefasst werden können.

Dieselben Aussagen treffen auch auf die Anfrage A.3 zu, wobei hier das Verbesserungspotential um einiges höher ist. Das ist damit zu begründen, dass es in den Datenmengen wesentlich mehr Aussagen über die Eigenschaft `ub:takesCourse` von Ressourcen gibt, und zwar genau 21 489, als die Benennung von `ub:GraduateStudent`-Ressourcen. Deshalb müssen im unoptimierten Fall 21 489 Folge-Anfragen gestellt werden, während es bei der Anfrage A.1, wie erwähnt, nur 1 874 sind.

Die Anfrage A.4 repräsentiert einen Sonderfall, da die beiden enthaltenen Tripelmuster gleich sind. Diese Tatsache wird im optimierten Fall bei der Restrukturierung entdeckt, da nach der Verschmelzung die MergeJoinedGPOs-Instanz eine RemoveEqualTriplePatterns-Instanz (siehe Definition 4.14 auf Seite 63) für den verschmolzenen Graphmuster-Operator als mögliche Folgeregel (siehe Tabelle 6.1 auf Seite 103) zurückgibt und diese tatsächlich ausgeführt werden kann. Ihre Ausführung führt dann zur Entfernung von einem der beiden Tripelmuster. Im unoptimierten Fall werden analog zu den Anfragen A.1 bis A.3 zunächst mittels einer SQL-Anfrage alle Möglichkeiten zur Übereinstimmung der Variablen `?s` und `?n` bestimmt und danach für jede Möglichkeit eine weitere Folge-Anfrage ausgeführt.

Auch bei den Anfragen A.5 und A.6 ist das Verbesserungspotential der Restrukturierung beachtlich. Dass hier im optimierten Fall die Ausführungszeit im Vergleich zu den Anfragen A.1 bis A.4 sogar noch stärker verkürzt werden kann, liegt an der größeren Menge von Tripelmustern, welche durch den Fast-Path-Algorithmus alle jeweils in einer Lösungsstufe zusammengefasst und mittels einer SQL-Anfrage gelöst werden können. Im unoptimierten Fall erhöht sich dagegen die Anzahl der auszuführenden Folge-Datenbankanfragen mit jedem zusätzlichen Tripelmuster. Im Vergleich zur Anfrage A.2, deren Lösung wie oben erwähnt $(1 + 1\,874)$ Datenbankanfragen benötigt, sind es bei Anfrage A.5 bereits $(1 + 1\,874 + c \cdot 1\,874) = (1 + (c + 1) \cdot 1\,874)$, wenn man davon ausgeht, dass jede der für das erste Tripelmuster gefundenen 1 874 `ub:GraduateStudent`-Ressourcen durchschnittlich c unterschiedliche `ub:takesCourse`-Eigenschaften hat.

Für die Anfrage A.7 kann durch die Restrukturierung keine Verkürzung der Ausführungszeit erreicht werden. Sie wird teilweise soger ein wenig verlängert. Dieses Verhalten ist auf die bereits erwähnte 1-zu-n-Beziehung zwischen den Ressourcen vom Typ `ub:GraduateStudent` und `ub:GraduateCourse`. Während der Restrukturierung werden die beiden Graphmuster-Operatoren mit dem ersten und dem dritten Tripelmuster verschmolzen, was dem Fast-Path-Algorithmus während der Planerzeugung die Möglichkeit gibt, diese Muster in einer Lösungsstufe zusammenzufassen. Während der Ausführung werden für diese Lösungsstufe mit einer SQL-Anfrage ungefähr $(c \cdot 1\,874)$ verschiedene Lösungen gefunden. Der Graphmuster-Operator, welcher für das optionale Graphmuster erstellt wurde kann natürlich nicht verschmolzen werden. Deshalb muss für jede der $(c \cdot 1\,874)$ Teillösungen noch eine weitere SQL-Anfrage zur jeweiligen Bestimmung der Variable `?n` ausgeführt werden. Damit ergeben sich letztendlich $(c \cdot 1\,874 + 1)$ Datenbankanfragen. Im unoptimierten Fall werden zunächst 1 874 `ub:GraduateStudent`-Ressourcen als Lösung für das erste Tripelmuster gefunden. Danach wird für jede dieser Teillösungen eine Folge-Datenbankanfrage zur Bindung der Variable `?n` ausgeführt. Da wie oben erwähnt jede `ub:GraduateStudent`-Ressource genau eine `ub:name`-Eigenschaft besitzt, vergrößert sich hierbei jedoch die Menge der Teillösungen nicht. Im letzten Schritt werden also noch einmal 1 874 SQL-Anfragen ausgeführt, um mit dem dritten Tripelmuster jeweils die möglichen Bindungen für die Variable `?c` zu bestimmen. Damit werden im unoptimierten Fall genau $(2 \cdot 1\,874 + 1)$ Datenbankanfragen ausgeführt, so dass bei $c > 2$ die Heuristik 6.1 keine Gültigkeit mehr besitzt.

Bei der Anfrage A.8 bewirkt die Anwendung der Heuristik wieder eine Verbesserung der Ausführungszeit, da hier im Gegensatz zur Anfrage A.7 nicht das Problem der 1-zu-n-Beziehung besteht. Das Verbesserungspotential ist hier jedoch nicht so hoch wie bei den Anfragen A.1 bis A.6, weil trotz der Restrukturierung Folge-Datenbankanfragen für den nicht verschmelzbaren Graphmuster-Operator des optionalen Graphmusters ausgeführt werden müssen. Dasselbe trifft auf die Anfrage A.9 zu.

In der Tabelle 6.2 ist weiterhin zu erkennen, dass bei einigen Anfragen das Verhältnis r_u der Anfrageausführungszeiten $t^*_{\text{exec}_u}$ und t_{exec_u} mit der Größe der Datenmenge zunächst abnimmt und dann stagniert, weil der jeweilige Anstieg von $t^*_{\text{exec}_u}$ und auch von t_{exec_u} zunächst abnimmt. Das gilt insbesondere für die Anfragen A.1 bis A.3 und A.6. Ein nachweisbarer Grund konnte dafür nicht gefunden werden. Es wird jedoch vermutet, dass entweder der Aufwand zur Ausführung der jeweils ersten SQL-Anfrage und der Übertragung der dabei ermittelten Daten oder die darauffolgende Erzeugung der Teillösungen innerhalb von ARQ nicht linear mit der Größe der Datenmenge steigt.

6.3.6 Evaluierung

Aufgrund der Erkenntnisse, welche durch die im vorherigen Abschnitt beschriebenen Tests und Messwerte gesammelt wurden, kann festgestellt werden, dass die Heuristik 6.1 in den meisten Fällen gilt. Zur Bestimmung der Ausnahmen wie beispielsweise Anfrage A.7 ist zusätzliches Wissen über die abgefragte Datenmenge nötig. Demzufolge wäre es möglich, eine erweiterte Variante der Heuristik 6.1 zu formulieren, welche unter Verwendung von Schema-Kenntnissen einige SQGMs oder sogar nur spezielle Teile eines SQGMs von der Anwendung der Restrukturierungsregel MergeAllJoinedGPOs ausschließt.

6.4 Ausnutzung von ARQ's Fast-Path-Algorithmus für alle gruppierten Tripelmuster

Die im vorherigen Abschnitt formulierte und untersuchte Heuristik 6.1 beschränkt sich auf SQGMs, deren Graphmuster-Operatoren lediglich Tripelmuster enthalten und diese Muster müssen die Eigenschaft haben, während der Ausführung des Fast-Path-Algorithmus in Lösungsstufen zusammengefasst werden zu können. Auf Basis der Kenntnis des Fast-Path-Algorithmus kann vermutet werden, dass auch Graphmuster-Operatoren, deren Tripelmuster sich nicht in einer Lösungsstufe zusammenfassen lassen, verschmolzen werden können, ohne dass sich die Ausführungszeit erhöht. Aufgrund dieser Vermutung kann die Beschränkung der Heuristik 6.1 aufgehoben werden.

Heuristik 6.2 Es wird angenommen, dass für die meisten SPARQL-Anfragen, welche durch ein den Voraussetzungen entsprechendes SQGM repräsentiert werden können, die Ausführung unter den vorausgesetzen Bedingungen schneller durchgeführt werden kann, wenn das SQGM mit Hilfe der angegebenen Restrukturierungsregel restrukturiert wird.
Effizienzkriterium: Ausführungszeit
Restrukturierungsregeln: MergeAllJoinedGPOs
Voraussetzungen:

- Die Anfrage wird mit der Standard-Anfragebearbeitungskomponente von ARQ ausgeführt.
- Die Anfrage wird direkt über einer persistent gespeicherten RDF-Datenmenge ausgeführt.
- Das SQGM enthält mehrere über Verbund-Operatoren oder Bäume von Verbund-Operatoren miteinander verbundene Graphmuster-Operatoren.
- Alle Graphmuster-Operatoren des SQGM enthalten lediglich Tripelmuster. □

6.4.1 Experimentelle Untersuchung

Um die Heuristik 6.2 untersuchen zu können, wurden einige der Anfragen A.1 bis A.9, welche für die Untersuchung der Heuristik 6.1 entwickelt wurden, angepasst. Dabei wurde in jeweils einem Tripelmuster an der Stelle des Prädikats die fest vorgegebene URI durch eine Variable ersetzt, um dafür zu sorgen, dass die Tripelmuster nicht mehr in einer Lösungsstufe zusammengefasst werden können.

In den Anfragen A.1, A.2, A.7 und A.8 wurde jeweils die URI *rdf:type* durch die Variable *?t* ersetzt und so die Anfragen A.10 bis A.13 formuliert. Die jeweils umformulierten Anfragen sind natürlich nicht mehr semantisch äquivalent zur jeweils ursprünglichen Version. Da aber in den betrachteten Datenmengen UnivBench(1,0), UnivBench(5,0) und UnivBench(10,0) alle Tripel mit der URI *ub:GraduateStudent* als Objekt immer das Prädikat *rdf:type* besitzen, liefern die umformulierten Anfragen über den betrachteten Datenmengen jeweils dieselben Ergebnisse für die Variablen *?s*, *?n* und *?c* wie ihre ursprünglichen Versionen.

Die Anfragen A.14 und A.15 sind aus den Anfragen A.9 und A.6 entstanden, wobei jeweils die URI *ub:takesCourse* durch die Variable *?tc* ersetzt wurde. Auch in diesem Fall sind die Anfragen jeweils nicht semantisch äquivalent, liefern aber ebenfalls dieselben Ergebnisse für die Variablen *?s*, *?n*, *?c* und *?e*.

Für die Testanfragen wurden analog zur Untersuchung der Heuristik 6.1 wieder jeweils zwei Testfälle erstellt und diese mit Hilfe der in Abschnitt 5.6 vorgestellten Testumgebung auf dem in Abschnitt 6.2 (Seite 91) beschriebenen Testsystem für die drei Datenmengen UnivBench(1,0), UnivBench(5,0) und UnivBench(10,0) ausgeführt. Die ermittelten Messwerte sind in der Tabelle 6.4 (Seite 112) zusammengefasst, wobei nur die jeweiligen Anfrageausführungszeiten t_{exec_u} aufgelistet sind. Die Initialisierungzeiten t_{init_u} und Planerzeugungszeiten t_{create_u} werden nicht dargestellt, weil sie, wie bei der Untersuchung der Heuristik 6.1 (siehe Tabelle 6.2 auf Seite 106, im Vergleich zur Anfrageausführungszeit t_{exec_u} vernachlässigbar gering sind.

Die Werte in den $t^*_{\text{exec}_u}$-Spalten entsprechen wieder den gemessenen Zeiten bei der Anfrageausführung ohne Normalisierung, Vereinfachung und Restrukturierung über der Datenmenge UnivBench(u,0) und die Werte in den t_{exec_u}-Spalten den Zeiten bei der Anfrageausführung mit Normalisierung, Vereinfachung und Restrukturierung. Analog enthalten die r_u-Spalten wieder das Verhältnis der entsprechenden Anfrageausführungszeiten $t^*_{\text{exec}_u}$ und t_{exec_u}.

Die Messwerte für alle Testanfragen bis auf A.10 zeigen, dass die Restrukturierung wie vermutet die Zeit der Anfrageausführung nicht erhöht. Sie sorgt teilweise sogar für eine Verminderung, wofür jedoch kein Grund angegeben werden kann.

Die Anfrage A.12 ist die einzigste, bei der eine Verschlechterung der Anfrageausführungszeit nach Anwendung der Restrukturierung beobachtet wurde. Sie ist aus der Anfrage A.7 entstanden, für die während der Untersuchung der Heuristik 6.1 ebenfalls eine Verschlechterung festgestellt wurde. Im Rahmen der Diskussion dieser Untersuchung (siehe Abschnitt 6.3.5 ab Seite 104) wurde die 1-zu-n-Beziehung zwischen den `ub:GraduateStudent`- und `ub:GraduateCourse`-Ressourcen als Grund für dieses Verhalten festgestellt. Auf die gleiche Weise wird die Verschlechterung der Ausführungszeit bei der Anfrage A.12 verursacht.

6.4.2 Evaluierung

Mit Hilfe der im vorherigen Abschnitt beschriebenen Tests und Messwerte kann festgestellt werden, dass die Heuristik 6.2 in den meisten Fällen gilt. Für eine spezielle Behandlung der Ausnahmen wie Anfrage A.12, könnten die bereits in Abschnitt 6.3.6 (Seite 108) vorgeschlagenen Erweiterung in Betracht gezogen werden.

6.5 Ausnutzung von ARQ's Fast-Path-Algorithmus bei beschränkten Tripelmustergruppen

Die Heuristik 6.2, welche im vorherigen Abschnitt formuliert und untersucht wurde, beschränkt sich auf SQGMs, deren Graphmuster-Operatoren lediglich Tripelmuster enthalten. In diesem Abschnitt wird darauf aufbauend eine Heuristik formuliert und untersucht, welche auch Wertebeschränkungen in den Graphmuster-Operatoren zulässt. Es wird vermutet, dass Graphmuster-Operatoren mit Wertebeschränkungen verschmolzen werden können, ohne dass sich die Ausführungszeit erhöht. Unter Umständen könnten solche Verschmelzungen die Ausführungszeit sogar verkürzen. Es sind Fälle denkbar, in denen durch die Einschränkung der Lösungsmenge eines verschmolzenen Graphmuster-Operators in anderen abzuarbeitenden Graphmuster-Operatoren die mögliche Lösungsmenge bereits kleiner ist und demzufolge beispielsweise weniger Folge-Datenbankanfragen notwendig sind.

Heuristik 6.3 Es wird angenommen, dass für die meisten SPARQL-Anfragen, welche durch ein den Voraussetzungen entsprechendes SQGM repräsentiert werden können, die Ausführung unter den vorausgesetzen Bedingungen schneller durchgeführt werden kann, wenn das SQGM mit Hilfe der angegebenen Restrukturierungsregel restrukturiert wird.
Effizienzkriterium: Ausführungszeit
Restrukturierungsregeln: MergeAllJoinedGPOs
Voraussetzungen:

- Die Anfrage wird mit der Standard-Anfragebearbeitungskomponente von ARQ ausgeführt.
- Die Anfrage wird direkt über einer persistent gespeicherten RDF-Datenmenge ausgeführt.
- Das SQGM enthält mehrere über Verbund-Operatoren oder Bäume von Verbund-Operatoren miteinander verbundene Graphmuster-Operatoren. □

6.5.1 Experimentelle Untersuchung

Zur Untersuchung der Heuristik 6.3 wurden einige der SPARQL-Anfragen, die im Rahmen der Evaluation der beiden Heuristiken 6.1 und 6.2 formuliert wurden, um zusätzliche FILTER-Klauseln erweitert, welche zur Erzeugung von Graphmuster-Operatoren mit Wertebeschränkungen führen. Diese FILTER-Klauseln beschänken die Werte der `ub:name`- oder `ub:takesCourse`-Eigenschaften, welche während den jeweiligen Musterübereinstimmungen gefunden wurden. Die entsprechend erweiterten Anfragen A.16 bis A.29 sind im Anhang A auf den Seiten A.16 bis A.29 abgebildet.

Wie bei den bereits beschriebenen Tests wurden jeweils zwei Testfälle für jede Testanfrage erstellt und abgearbeitet. In Tabelle 6.5 (Seite 112) sind die ermittelten Messwerte analog zur Untersuchung der Heuristik 6.2 in Abschnitt 6.4.1 (Seite 109) aufgelistet.

Die Anfragen A.16 bis A.19 sind Varianten der Anfragen A.1 bis A.3, welche um ein oder zwei FILTER-Klauseln erweitert wurden. Bei den Anfragen A.16 und A.17 sind sich analog zu A.1 und A.2 (Tabelle 6.2 auf Seite 112) sowohl die Ausführungszeiten, als auch deren Verbesserungen ähnlich. Der Grund dafür ist derselbe. Im unoptimierten Fall ohne Restrukturierung findet die SQL-Anfrage für das jeweils erste Tripelmuster eine bestimmte Anzahl an `ub:GraduateStudent`-Ressourcen für die Variable *?s* und es muss eine, dieser Anzahl entsprechende Menge weiterer SQL-Anfragen für das jeweils zweite Tripelmuster ausgeführt werden. Im Vergleich zur Untersuchung für die Heuristik 6.1 fällt für die Messwerte der Anfragen A.16 und A.17 jedoch auf, dass für sie das Verhältnis r_u im Gegensatz zu den Messwerten der Anfragen A.1 bzw. A.2 erst bei größeren Datenmengen abnimmt. Die Tabelle 6.3 zeigt die erwähnten Werte. Auch für dieses Verhalten konnte kein Grund gefunden werden.

Die semantisch äquivalenten Anfragen A.18 und A.19 können durch die Anwendung der Restrukturierungsregeln MergeAllJoinedGPOs schneller ausgeführt werden. Das Verbesserungspotential ist für die Anfrage A.18 jedoch höher. Wegen des syntaktischen Unterschieds werden

Anfrage	r_1	r_5	r_{10}
A.1 (Seite 124)	ca. 7,7	ca. 2,3	ca. 2,3
A.16 (Seite 125)	ca. 7,9	ca. 8,3	ca. 2,0
A.2 (Seite 124)	ca. 6,3	ca. 2,3	ca. 2,3
A.17 (Seite 125)	ca. 7,0	ca. 6,8	ca. 1,9

Tabelle 6.3: Vergleich der Abnahme der jeweiligen Verhältnisse r_u bei den Anfragen A.16 und A.1 bzw. A.17 und A.2

zwei unterschiedliche SQGMs erzeugt. Die restrukturierten SQGMs sind jedoch für beide Anfragen gleich. Deshalb sind auch die Ausführungszeiten im optimierten Fall annähernd gleich. Die Unterschiede im Verbesserungspotential sind auf die unterschiedlichen Ausführungszeiten im unoptimierten Fall zurückzuführen. Das liegt daran, dass bei der Ausführung der Anfrage A.19 nach der Bestimmung der Teillösung für das erste Tripelmuster bereits eine Einschränkung der Lösungsmenge durch die erste FILTER-Klausel bewirkt wird. Dadurch müssen zur Bestimmung der jeweiligen Lösungen des zweiten Tripelmusters weniger Folge-Datenbankanfragen ausgeführt werden als bei Anfrage A.18.

Die Anfragen A.20 bis A.23 sind Varianten der Anfrage A.5 und ihre Ausführungszeit wird durch Anwendung der Heuristik deutlich verkürzt. Während das Verbesserungspotential der Anfragen A.21 in etwa dem der Anfrage A.5 entspricht, ist es bei A.20 und A.22 etwas geringer. Das liegt wie bei der Anfrage A.19 daran, dass im unoptimierten Fall die Menge der Teillösungen bereits eingeschränkt wird.

Die Anfrage A.23 repräsentiert einen Sonderfall. Sie hat eine leere Ergebnismenge, weil sich die enthaltenen Wertebeschränkungen widersprechen. Im unoptimierten Fall wird der Widerspruch jedoch nicht erkannt und die Anfrage wie gewohnt ausgeführt. Erst bei der Lösungsmengen-Einschränkung auf Basis der Wertebeschränkungen werden alle gefundenen Teillösungen verworfen. Im optimierten Fall hingegen, wird der Widerspruch im verschmolzenen Graphmuster-Operator entdeckt, da für die Verschmelzungsregel MergeJoinedGPOs eine, den verschmolzenen Operator betreffende FindContradiction-Instanz (siehe Definition 4.19 auf Seite 65) als mögliche Folgeregel (siehe Tabelle 6.1 auf Seite 103) abgearbeitet wird. Dabei wird der verschmolzene Graphmuster-Operator als widersprüchlich gekennzeichnet und dann mit Hilfe der Regeln zum Behandeln von Widersprüchen (siehe Abschnitt 4.3.2 ab Seite 65) im SQGM nach oben propagiert und schließlich entfernt. So kann während der Ausführung sofort erkannt werden, dass die Anfrage eine leere Ergebnismenge hat und die Ausführungszeit wird drastisch reduziert.

Die restlichen getesteten Anfragen enthalten teilweise optionale Graphmuster oder Tripelmuster, welche durch den Fast-Path-Algorithmus nicht in eine gemeinsame Lösungsstufe aufgenommen werden können. Die Messergebnisse zeigen für diese Anfragen, dass ihre Ausführungszeiten durch Anwendung der Heuristik 6.3 verkürzt werden. Besonders erwähnenswert sind die Anfragen A.28 und A.29. Sie enthalten optionale Graphmuster und die Tripelmuster können vom Fast-Path-Algorithmus nicht in gemeinsamen Lösungsstufen zusammengefasst werden. Hier wird die Verkürzung der Ausführungszeiten im optimierten Fall nur durch die Einschränkung der Menge von Teillösungen erreicht, welche erst durch die Verschmelzung des Graphmuster-Operators mit der Wertebeschränkung mit dem Graphmuster-Operator mit den ersten beiden Tripelmustern ermöglicht wird.

6.5.2 Evaluierung

Durch die im vorherigen Abschnitt beschriebenen Tests konnte die Vermutung, welche zur Heuristik 6.3 geführt hat, bestätigt werden. Im Gegensatz zu den Heuristiken 6.1 und 6.2 formuliert diese Heuristik keine Einschränkungen bezüglich der Graphmuster-Operatoren eines SQGM. Deshalb kann sie für alle SQGMs, die mehrere über Verbund-Operatoren oder Bäume von Verbund-Operatoren miteinander verbundene Graphmuster-Operatoren enthalten, angewandt werden und führt bei einer Anfrage an eine persistent gespeicherte Datenmenge in den meisten Fällen zu einer Verkürzung der Ausführungszeit.

Anfrage	$t^*_{\text{exec}_1}$	t_{exec_1}	r_1	$t^*_{\text{exec}_5}$	t_{exec_5}	r_5	$t^*_{\text{exec}_{10}}$	$t_{\text{exec}_{10}}$	r_{10}
A.10 (Seite 125)	2 108 ms	1 136 ms	ca. 1,9	14 476 ms	7 855 ms	ca. 1,8	33 658 ms	16 203 ms	ca. 2,1
A.11 (Seite 125)	2 189 ms	1 211 ms	ca. 1,8	15 007 ms	8 290 ms	ca. 1,8	30 155 ms	16 977 ms	ca. 1,8
A.12 (Seite 125)	4 108 ms	5 237 ms	ca. 0,8	26 795 ms	34 745 ms	ca. 0,8	56 168 ms	71 483 ms	ca. 0,8
A.13 (Seite 125)	5 925 ms	3 328 ms	ca. 1,8	38 987 ms	22 551 ms	ca. 1,7	80 283 ms	46 233 ms	ca. 1,7
A.14 (Seite 125)	895 914 ms	890 748 ms	ca. 1,0	?	?		?	?	
A.15 (Seite 125)	895 567 ms	884 288 ms	ca. 1,0	?	?		?	?	

Tabelle 6.4: Messwerte der Tests für die Heuristik 6.2

Anfrage	$t^*_{\text{exec}_1}$	t_{exec_1}	r_1	$t^*_{\text{exec}_5}$	t_{exec_5}	r_5	$t^*_{\text{exec}_{10}}$	$t_{\text{exec}_{10}}$	r_{10}
A.16 (Seite 125)	2 191 ms	276 ms	ca. 7,9	14 893 ms	1 803 ms	ca. 8,3	44 887 ms	22 125 ms	ca. 2,0
A.17 (Seite 125)	2 278 ms	327 ms	ca. 7,0	15 620 ms	2 288 ms	ca. 6,8	45 004 ms	23 234 ms	ca. 1,9
A.18 (Seite 126)	10 004 ms	2 029 ms	ca. 4,9	100 105 ms	32 488 ms	ca. 3,1	197 422 ms	65 140 ms	ca. 3,0
A.19 (Seite 126)	24 306 ms	1 970 ms	ca. 12,3	173 488 ms	32 763 ms	ca. 5,3	353 058 ms	64 420 ms	ca. 5,5
A.20 (Seite 126)	3 603 ms	365 ms	ca. 9,9	26 721 ms	2 552 ms	ca. 10,5	75 356 ms	23 421 ms	ca. 3,2
A.21 (Seite 126)	5 972 ms	367 ms	ca. 16,2	39 517 ms	2 546 ms	ca. 15,5	81 575 ms	5 051 ms	ca. 16,2
A.22 (Seite 126)	3 608 ms	368 ms	ca. 9,8	27 246 ms	2 566 ms	ca. 10,6	53 730 ms	5 086 ms	ca. 10,6
A.23 (Seite 126)	1 703 ms	$<$ 0 ms	$\rightarrow \infty$	9 500 ms	$<$ 0 ms	$\rightarrow \infty$	20 985 ms	$<$ 0 ms	$\rightarrow \infty$
A.24 (Seite 126)	6 124 ms	1 975 ms	ca. 3,1	40 639 ms	15 988 ms	ca. 2,5	83 451 ms	29 885 ms	ca. 2,8
A.25 (Seite 126)	4 203 ms	1 111 ms	ca. 3,8	28 139 ms	7 782 ms	ca. 3,6	57 859 ms	15 436 ms	ca. 3,7
A.26 (Seite 127)	2 156 ms	1 147 ms	ca. 1,9	16 999 ms	8 023 ms	ca. 2,1	52 928 ms	40 269 ms	ca. 1,3
A.27 (Seite 127)	2 247 ms	1 212 ms	ca. 1,9	15 129 ms	8 642 ms	ca. 1,8	53 938 ms	41 355 ms	ca. 1,3
A.28 (Seite 127)	6 085 ms	2 761 ms	ca. 2,2	40 458 ms	22 187 ms	ca. 1,8	82 687 ms	42 981 ms	ca. 1,9
A.29 (Seite 127)	4 182 ms	1 950 ms	ca. 2,1	27 939 ms	14 178 ms	ca. 2,0	57 502 ms	28 612 ms	ca. 2,0

Tabelle 6.5: Messwerte der Tests für die Heuristik 6.3

Legende:
$t^*_{\text{exec}_u}$ – repräsentiert die jeweils für die Datenmenge UnivBench(u,0) gemessene Anfrageausführungszeit bei Bearbeitung ohne Anwendung der Restrukturierungsregeln
t_{exec_u} – repräsentiert die jeweils für die Datenmenge UnivBench(u,0) gemessene Anfrageausführungszeit bei Bearbeitung mit Anwendung der Restrukturierungsregeln
r_u – diese Spalten enthalten die jeweiligen Verhältnisse $t^*_{x_u}/t_{x_u}$ der Anfrageausführungszeiten
? – bei der Ermittlung dieser Werte wurde die Messung nach ca. 8 Stunden abgebrochen

6.6 Ausnutzung von ARQ's Fast-Path-Algorithmus bei vereinigten Graphmustern

In diesem Abschnitt wird eine Heuristik formuliert und untersucht, welche sich auf SPARQL-Anfragen bezieht, die von der ARQ-Anfragebearbeitungskomponente über einer persistent gespeicherten Datenmenge ausgeführt werden und gruppierte Graphmuster-Vereinigung enthalten.

6.6.1 Zugrundliegende Vermutung

Im Rahmen der Untersuchung der Heuristiken 6.1 bis 6.3 wurde festgestellt, dass ein Zusammenfassen von gruppierten einfachen Graphmustern in den meisten Fällen eine schnellere Anfrageausführung ermöglicht. Wenn sich beliebige Graphmuster zusammen mit einer Graphmuster-Vereinigung (siehe Abschnitt 2.3.5 auf Seite 11) innerhalb einer Graphmuster-Gruppe befinden, so könnten diese Graphmuster in die Vereinigung verschoben werden. Das schafft unter Umständen weitere Möglichkeiten für die Zusammenfassung von einfachen Graphmustern und führt demzufolge in den meisten Fällen ebenfalls zu einer Verkürzung der Anfrageausführungszeit. Die Verschiebung eines Graphmusters in eine Graphmuster-Vereinigung bedeutet, dass das Graphmuster mit jedem der vereinigten Graphmuster in jeweils einer Graphmuster-Gruppe zusammengefasst wird. Diese neuen Graphmuster-Gruppen bilden dann die neuen vereinigten Graphmuster der Graphmuster-Vereinigung.

Beispiel 6.3 *Das Graphmuster der folgenden SPARQL-Anfrage ist eine Graphmuster-Gruppe, welche eine Graphmuster-Vereinigung und ein einfaches Graphmuster gruppiert.*

```
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?n
WHERE {
   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }

   ?s ub:name ?n .
}
```

Das einfache Graphmuster kann in die Vereinigun verschoben werden. Dabei wird es jeweils mit den vereinigten einfachen Graphmustern in einer Graphmuster-Gruppe zusammengefasst. Es entsteht die folgende, semantisch äquivalente Anfrage.

```
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?n
WHERE {
   {
      { ?s rdf:type ub:GraduateStudent }
      { ?s ub:name ?n }
   }
   UNION
   {
      { ?s rdf:type ub:UndergraduateStudent }
      { ?s ub:name ?n }
   }
}
```

Die gruppierten einfachen Graphmuster innerhalb der Vereinigung können jeweils zu einem einfachen Graphmuster verschmolzen werden, wodurch die folgende, ebenfalls semantisch äquivalente Anfrage entsteht.

```
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?n
WHERE {
   {
      ?s rdf:type ub:GraduateStudent .
      ?s ub:name ?n .
   }
   UNION
   {
      ?s rdf:type ub:UndergraduateStudent .
      ?s ub:name ?n .
   }
}
```

Bei einer Ausführung beider Anfragen durch die ARQ-Anfragebearbeitungskomponente über einer persistenten Datenmenge wird vermutet, dass die dritte Anfrage schneller ausgeführt werden kann, weil der Fast-Path-Algorithmus in diesem Fall die Musterübereinstimmung für beide Tripelmuster-Paare in jeweils einer Lösungsstufe verbinden kann. □

Die Heuristik setzt natürlich voraus, dass eine Anfrage Graphmuster-Vereinigungen enthält, welche in Graphmuster-Gruppen mit anderen Graphmustern verbunden sind. Die Aufgabe bei der Anwendung der Heuristik besteht darin, die gruppierten Graphmuster in die Graphmuster-Vereinigungen zur verschieben. Im Folgenden wird die Umsetzung der skizzierten Aufgabe für diese Heuristik auf Basis des SQGM beschrieben.

6.6.2 Notwendige Transformationsregeln

In einem SQGM werden Graphmuster-Vereinigungen durch Vereinigungs-Operatoren (siehe Definition 3.13 auf Seite 25) oder Bäume von Vereinigungs-Operatoren modelliert. Graphmuster-Vereinigungen innerhalb von Graphmuster-Gruppen werden im SQGM durch Vereinigungs-Operatoren, deren Ausgangs-Variablendatenfluss ein Eingangs-Datenfluss eines Verbund-Operators ist, repräsentiert. Der Operator, der den anderen Eingangs-Datenfluss des Verbund-Operators bereitstellt, modelliert das Graphmuster, welches mit der Graphmuster-Vereinigung in derselben Graphmuster-Gruppe verbunden ist. Um dieses Graphmuster in die Graphmuster-Vereinigung zu verschieben, muss der Vereinigungs-Operator im SQGM die Position mit dem Verbund-Operators tauschen, wobei der Verbund-Operator und der andere Operator dupliziert werden. Die folgenden beiden Transformationsregeln können hierfür verwendet werden.

Transformationsregel 6.5 – PullJoinedUnionLeft

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{left}}.\text{optional} \neq \mathsf{T}$
ii) $o_{\text{left}} \in UO$
iii) $|o_{\text{left}}.\text{output}| = 1$

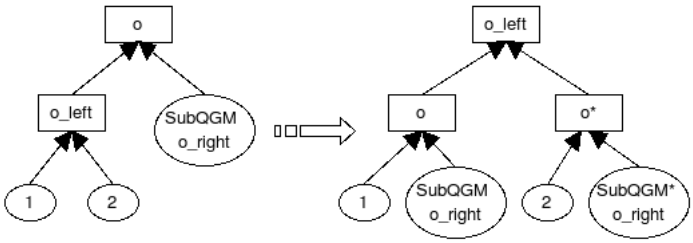
Anwendung: Seien $(j_1, j_2) := o_{\text{left}}.\text{input}$. Sei weiterhin $SubSQGM_{o_{\text{right}}}$ der Teilbaum des SQGM, welcher den Operator o_{right} als Wurzel hat und sei $SubSQGM^*_{o_{\text{right}}}$ eine Kopie davon mit o^*_{right} als Wurzel.

Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt

i) $o_{\text{left}}.\text{output} := o.\text{output}$
ii) $i^*_{\text{right}}.\text{vars} := i_{\text{right}}.\text{vars}$
iii) $i^*_{\text{right}}.\text{optional} := i_{\text{right}}.\text{optional}$
iv) $o^*_{\text{right}}.\text{output} := (i^*_{\text{right}})$
v) $o^*.\text{input} := (j_2, i^*_{\text{right}})$
vi) $o^*.\text{provVars} := j_2.\text{vars} \cup i_{\text{right}}.\text{vars}$
vii) $o^*.\text{output} := (i^*_{\text{left}})$
viii) $i^*_{\text{left}}.\text{vars} := j_2.\text{vars} \cup i_{\text{right}}.\text{vars}$
ix) $i^*_{\text{left}}.\text{optional} := \mathsf{F}$
x) $o.\text{input} := (j_1, i_{\text{right}})$
xi) $o.\text{provVars} := j_1.\text{vars} \cup i_{\text{right}}.\text{vars}$
xii) $o.\text{output} := (i_{\text{left}})$
xiii) $i_{\text{left}}.\text{vars} := j_1.\text{vars} \cup i_{\text{right}}.\text{vars}$
xiv) $o_{\text{left}}.\text{input} := (i_{\text{left}}, i^*_{\text{left}})$
xv) $o_{\text{left}}.\text{provVars} := j_1.\text{vars} \cup j_2.\text{vars} \cup i_{\text{right}}.\text{vars}$
xvi) $OP' := OP \cup \{o^*\}$
xvii) $DF' := DF \cup \{i^*_{\text{left}}, i^*_{\text{right}}\}$

wobei $o^* \in JO$ ein neu zu erzeugender Verbund-Operator und $i^*_{\text{left}} \in VF$ und $i^*_{\text{right}} \in VF$ neu zu erzeugende Variablendatenflüsse sind.

Erläuterung: Wenn für einen Verbund-Operator o der linke Eingangsdatenfluss i_{left} nicht optional ist und von einem Vereinigungs-Operator o_{left} ohne weitere Ausgangsdatenflüsse bereitgestellt wird, dann vertauscht diese Regel die Position von o_{left} und o innerhalb des SQGM. Dabei werden von o und dem Teilbaum $SubSQGM_{o_{\text{right}}}$, welcher o's rechten Eingangsdatenfluss i_{right} bereitstellt, Kopien o^* und $SubSQGM^*_{o_{\text{right}}}$ angelegt. Die Verbund-Operatoren o und o^* stellen die neuen Eingänge zum Vereinigungs-Operator o_{left} bereit. Ihre neuen Eingänge werden neben den Ausgängen von $SubSQGM_{o_{\text{right}}}$ bzw. $SubSQGM^*_{o_{\text{right}}}$ noch die ursprünglichen Eingänge j_1 bzw. j_2 des Vereinigungs-Operators. Die folgende Abbildung deutet diese Transformation an.



Begründung: Der Verbund-Operator o repräsentiert eine Gruppe g von zwei Graphmustern. Eines davon wird durch o_{left} repräsentiert und entspricht einer Graphmuster-Vereinigung u. Durch die Voraussetzung i) wird ausgeschlossen, dass diese zweite Gruppe ein optionales Graphmuster ist. Seien a und b die durch u vereinigten Graphmuster. Das andere Graphmuster p der Gruppe wird durch o_{right} bzw. den Teilbaum $SubSQGM_{o_{\text{right}}}$ repräsentiert. Der kopierte Teilbaum $SubSQGM^*_{o_{\text{right}}}$ repräsentiert ebenfalls das Graphmuster p.

Durch die Transformation wird a in einer Graphmuster-Gruppe g_1 mit p und b in einer weiteren Graphmuster-Gruppe g_2 mit p verbunden und diese beiden Graphmuster-Gruppen bilden die neuen durch u vereinigten Graphmuster. Nach der Transformation ersetzt diese Graphmuster-Vereinigung die ursprüngliche Graphmuster-Gruppe g.

Sei s eine Lösung der durch o vor der Transformation repräsentierten Graphmuster-Gruppe. Dann ist s auch Lösung von u und von p. Da s Lösung von u ist, ist s weiterhin Lösung von a oder b. Ohne Beschränkung der Allgemeinheit sei s Lösung von a. Nach der Transformation ist s natürlich immer noch Lösung von a und p. Daraus folgt, dass s Lösung von g_1 ist und somit auch Lösung von u nach der Transformation.
Sei andererseits s eine Lösung von u nach der Transformation. Damit ist s auch Lösung von g_1 oder g_2. Sei ohne Beschränkung der Allgemeinheit s Lösung von g_1. Dann ist s auch Lösung von a und p nach der Transformation. Da a und p durch die Transformation nicht verändert wurden, ist s auch Lösung von a und p vor der Transformation. Demzufolge ist s Lösung von u vor der Transformation und letztendlich auch von o.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt. □

Transformationsregel 6.6 – PullJoinedUnionRight

Definitionsbereich: $o \in JO \subset OP$

Voraussetzung: Seien $(i_{\text{left}}, i_{\text{right}}) := o.\text{input}$; $o_{\text{left}} := ProvOp(i_{\text{left}})$ und $o_{\text{right}} := ProvOp(i_{\text{right}})$. Es gelte:

i) $i_{\text{right}}.\text{optional} \neq \mathsf{T}$
ii) $o_{\text{right}} \in UO$
iii) $|o_{\text{right}}.\text{output}| = 1$

Anwendung: Seien $(j_1, j_2) := o_{\text{right}}.\text{input}$. Sei weiterhin $SubSQGM_{o_{\text{left}}}$ der Teilbaum des SQGM, welcher den Operator o_{left} als Wurzel hat und sei $SubSQGM^*_{o_{\text{left}}}$ eine Kopie davon mit o^*_{left} als Wurzel.

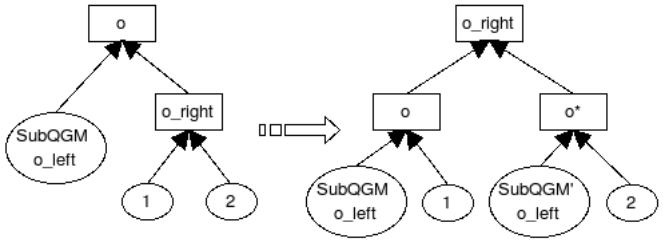
Aus $q = (OP, DF, r, dflt, NG)$ ergibt sich $q' := (OP', DF', r, dflt, NG)$ wie folgt

i) $o_{\text{right}}.\text{output} := o.\text{output}$
ii) $i^*_{\text{left}}.\text{vars} := i_{\text{left}}.\text{vars}$
iii) $i^*_{\text{left}}.\text{optional} := i_{\text{left}}.\text{optional}$
iv) $o^*_{\text{left}}.\text{output} := (i^*_{\text{left}})$
v) $o^*.\text{input} := (i^*_{\text{left}}, j_2)$
vi) $o^*.\text{provVars} := i_{\text{left}}.\text{vars} \cup j_2.\text{vars}$
vii) $o^*.\text{output} := (i^*_{\text{right}})$
viii) $i^*_{\text{right}}.\text{vars} := i_{\text{left}}.\text{vars} \cup j_2.\text{vars}$
ix) $i^*_{\text{right}}.\text{optional} := \mathsf{F}$
x) $o.\text{input} := (i_{\text{left}}, j_1)$
xi) $o.\text{provVars} := i_{\text{left}}.\text{vars} \cup j_1.\text{vars}$
xii) $o.\text{output} := (i_{\text{right}})$
xiii) $i_{\text{right}}.\text{vars} := i_{\text{left}}.\text{vars} \cup j_1.\text{vars}$
xiv) $o_{\text{right}}.\text{input} := (i_{\text{right}}, i^*_{\text{right}})$
xv) $o_{\text{right}}.\text{provVars} := j_1.\text{vars} \cup j_2.\text{vars} \cup i_{\text{left}}.\text{vars}$
xvi) $OP' := OP \cup \{o^*\}$
xvii) $DF' := DF \cup \{i^*_{\text{left}}, i^*_{\text{right}}\}$

wobei $o^* \in JO$ ein neu zu erzeugender Verbund-Operator und $i^*_{\text{left}} \in VF$ und $i^*_{\text{right}} \in VF$ neu zu erzeugende Variablendatenflüsse sind.

Abbhängigkeit: Diese Regel setzt voraus, dass das betrachtete SQGM die erste Eigenschaft der Normalform (siehe Definition 4.1 auf Seite 52) erfüllt.

Erläuterung: Diese Regel ist das Pendant zu PullJoinedUnionLeft bezogen auf den rechten Eingangsdatenfluss, wie die folgende Abbildung andeutet.



□

Die Tabelle 6.6 listet die möglichen Folgeregeln der beiden Transformationsregeln auf.

Ausgeführte Regel	Bedingung	mögliche Folgeregel	zu verw. Operator
PullJoinedUnionLeft	–	PropagateUnreqVarUnion	o_{left}
PullJoinedUnionRight	–	PropagateUnreqVarUnion	o_{right}

Tabelle 6.6: Mögliche Folgeregeln der Transformationsregeln des Abschnitts 6.6.2

6.6.3 Notwendige Restrukturierungsregeln

Auf Basis der Transformationsregeln PullJoinedUnionLeft und PullJoinedUnionRight, welche im vorherigen Abschnitt spezifiziert wurden, kann eine Restrukturierungsregel formuliert werden, die das Ziel der Verschiebung aller mit Graphmuster-Vereinigungen gruppierten Graphmuster in die Vereinigungen auf das SQGM überträgt.

Restrukturierungsregel 6.2 – PullAllJoinedUnions

Ziel: Diese Regel verfolgt das Ziel alle mit einem Vereinigungs-Operator verbundenen Operatoren mit den eingangs-bereitstellenden Operatoren des Vereinigungsoperators zu verbinden und damit die Vereinigungsoperatoren im SQGM so weit wie möglich in Richtung Wurzel zu propagieren.

Schritte: Zur Erreichung des Ziels müssen die Vereinigungs-Operatoren schrittweise jeweils um eine Position in Richtung Wurzel des SQGM propagiert werden.

Algorithmen: Um festzustellen, ob ein weiterer Restrukturierungsschritt für diese Regel ausführbar ist, muss lediglich festgestellt werden, ob im SQGM ein Vereinigungs-Operator exisitiert, für den eine PullJoinedUnionLeft- oder PullJoinedUnionRight-Instanz ausführbar ist. Kann ein solcher Vereinigungs-Operator bestimmt werden, so ist ein weiterer Schritt ausführbar und ansonsten nicht.
Zur Erstellung der initialen Sequenz von Transformationsregelinstanzen für die Ausführung von einem Schritt muss der Vereinigungs-Operator bestimmt werden, für den eine PullJoinedUnionLeft- bzw. PullJoinedUnionRight-Instanz ausführbar ist. Dieser Operator existiert, da sonst kein weiterer Schritt mehr ausführbar gewesen wäre. Die entsprechende Regelinstanz wird in die initiale Sequenz aufgenommen, womit deren Erstellung abgeschlossen ist. □

6.6.4 Spezifikation der Heuristik

Die in den vorherigen Abschnitten spezifizierten Regeln bilden die Grundlage zur Formulierung der Heuristik, welche auf den in Abschnitt 6.6.1 (auf Seite 113) dargelegten Vermutungen basiert.

Heuristik 6.4 Es wird angenommen, dass für die meisten SPARQL-Anfragen, welche durch ein den Voraussetzungen entsprechendes SQGM repräsentiert werden können, die Ausführung unter den vorausgesetzen Bedingungen schneller durchgeführt werden kann, wenn das SQGM mit Hilfe der angegebenen Restrukturierungsregel restrukturiert wird.
Effizienzkriterium: Ausführungszeit
Restrukturierungsregeln: PullAllJoinedUnions, MergeAllJoinedGPOs

Voraussetzungen:

- Die Anfrage wird mit der Standard-Anfragebearbeitungskomponente von ARQ ausgeführt.
- Die Anfrage wird direkt über einer persistent gespeicherten RDF-Datenmenge ausgeführt.
- Das SQGM enthält mindestens einen Verbund-Operator, der einen Vereinigungs-Operator mit einem beliebigen anderen Operator verbindet. □

6.6.5 Experimentelle Untersuchung

Zur Untersuchung der Heuristik 6.4 wurden die Testanfragen A.30 bis A.41 (siehe Anhang A) formuliert. Mit diesen Anfragen wurden analog zu den bereits beschriebenen Untersuchungen (siehe beispielsweise Abschnitt 6.3.5 ab Seite 104) Testfälle erstellt und diese ausgeführt. Die dabei ermittelten Messwerte sind in der Tabelle 6.7 (Seite 119) zusammengefasst.

Die Anfragen A.30 bis A.33 sind ähnlich. Bei allen entstehen durch die Restrukturierung zwei Graphmuster-Operatoren, welche jeweils zwei Tripelmuster enthalten. Beide Graphmuster-Operatoren stellen jeweils einen Eingangs-Datenfluss des in Richtung Wurzel propagierten Vereinigungs-Operators bereit. Die beiden Tripelmuster in den beiden durch Verschmelzung entstandenen Graphmuster-Operatoren können durch den Fast-Path-Algorithmus jeweils in einer Lösungsstufe zusammengefasst werden. Dadurch können diese Anfragen, wie vermutet, schneller ausgeführt werden.

Die Unterschiede im Verbesserungspotential zwischen den Anfragen A.30 und A.31 bzw. zwischen A.32 und A.33 liegen an den Unterschieden zwischen den jeweiligen Ausführungszeiten ohne Restrukturierung. Diese sind darauf zurückzuführen, dass die Datenmengen jeweils mehr Tripel mit dem Prädikat `ub:name` bzw. `ub:takesCourse` beinhalten als `ub:GraduateStudent`- und `ub:UndergraduateStudent`-Ressourcen und deshalb bei der Anfrage A.31 bzw. A.33 mehr Folge-SQL-Anfragen provoziert werden.

Bei den Anfragen A.34 und A.35 ergibt sich durch die Restrukturierung ebenfalls die Möglichkeit zur Gruppierung von Tripelmustern in gleichen Lösungsstufen und damit können auch diese Anfragen schneller ausgeführt werden. Die jeweilige Zeitersparnis bei der Anfrageausführung mit Optimierung gegenüber der Ausführung ohne ist hier jedoch nicht so hoch wie bei den Anfragen A.30 bis A.33. Das liegt daran, dass für das optionale Graphmuster in jedem Fall zusätzliche SQL-Anfragen ausgeführt werden müssen.

Die restlichen Anfragen A.36 bis A.41 entsprechen den bereits erwähnten mit dem Unterschied, dass die durch Restrukturierung zusammengeführten Tripelmuster nicht in gemeinsamen Lösungsstufen ausgewertet werden können. Diese Anfragen erinnern daher auch an die Testanfragen, welche für die Untersuchung der Heuristik 6.2 (siehe Abschnitt 6.4.1 ab Seite 109) formuliert wurden. Dort wurde beobachtet, dass die Anfrageausführungszeiten durch die Restrukturierung etwas verkürzt werden, wofür kein Grund angegeben werden konnte. Dasselbe trifft auch hier zu.

6.6.6 Evaluierung

Mit Hilfe der experimentellen Untersuchungen kann festgestellt werden, dass die Heuristik 6.4 gilt und unter den für sie angegebenen Voraussetzungen angewandt werden kann.

Anfrage	Messung	$t^*_{x_1}$	t_{x_1}	r_1	$t^*_{x_5}$	t_{x_5}	r_5	$t^*_{x_{10}}$	$t_{x_{10}}$	r_{10}
A.30 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	< 0 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	1 ms	< 0 ms		< 0 ms	< 0 ms		5 ms	1 ms	
	Anfrageausführungszeit t_{exec_u}	9 007 ms	1 018 ms	ca. 8,8	84 133 ms	31 587 ms	ca. 2,7	184 091 ms	67 628 ms	ca. 2,7
A.31 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		13 ms	< 0 ms		15 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	34 508 ms	1 009 ms	ca. 34,2	246 831 ms	29 593 ms	ca. 8,3	541 214 ms	66 724 ms	ca. 8,1
A.32 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		1 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		2 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	9 591 ms	1 396 ms	ca. 6,9	76 435 ms	10 058 ms	ca. 7,6	180 040 ms	71 179 ms	ca. 2,5
A.33 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	1 ms		10 ms	< 0 ms		10 ms	7 ms	
	Anfrageausführungszeit t_{exec_u}	45 937 ms	1 410 ms	ca. 32,6	314 589 ms	26 935 ms	ca. 11,7	658 162 ms	78 469 ms	ca. 8,4
A.34 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	< 0 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		8 ms	10 ms		3 ms	6 ms	
	Anfrageausführungszeit t_{exec_u}	31 095 ms	24 782 ms	ca. 1,3	223 773 ms	191 260 ms	ca. 1,2	461 750 ms	408 186 ms	ca. 1,1
A.35 (Seite 127)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	1 ms		?	?	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		5 ms	< 0 ms		?	?	
	Anfrageausführungszeit t_{exec_u}	67 891 ms	24 716 ms	ca. 2,7	458 640 ms	201 837 ms	ca. 2,3	?	?	
A.36 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	1 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	1 ms	< 0 ms		9 ms	< 0 ms		3 ms	1 ms	
	Anfrageausführungszeit t_{exec_u}	9 068 ms	4 752 ms	ca. 1,9	84 897 ms	59 745 ms	ca. 1,4	185 157 ms	135 274 ms	ca. 1,4
A.37 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	1 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		7 ms	< 0 ms		5 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	45 107 ms	27 082 ms	ca. 1,7	309 322 ms	225 500 ms	ca. 1,4	685 108 ms	465 826 ms	ca. 1,5
A.38 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	< 0 ms		< 0 ms	< 0 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		2 ms	< 0 ms		7 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	9 531 ms	5 171 ms	ca. 1,8	62 958 ms	33 938 ms	ca. 1,9	179 987 ms	125 266 ms	ca. 1,4
A.39 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	< 0 ms		< 0 ms	< 0 ms		6 ms	7 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		4 ms	1 ms		14 ms	< 0 ms	
	Anfrageausführungszeit t_{exec_u}	61 005 ms	36 325 ms	ca. 1,8	406 970 ms	274 797 ms	ca. 1,5	855 538 ms	575 238 ms	ca. 1,5
A.40 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	1 ms		< 0 ms	1 ms	
	Planerzeugungszeit t_{create_u}	< 0 ms	< 0 ms		10 ms	4 ms		3 ms	3 ms	
	Anfrageausführungszeit t_{exec_u}	31 117 ms	28 486 ms	ca. 1,1	221 856 ms	205 093 ms	ca. 1,1	463 193 ms	427 224 ms	ca. 1,1
A.41 (Seite 128)	Initialisierungszeit t_{init_u}	< 0 ms	1 ms		< 0 ms	1 ms		?	?	
	Planerzeugungszeit t_{create_u}	4 ms	6 ms		12 ms	3 ms		?	?	
	Anfrageausführungszeit t_{exec_u}	83 254 ms	64 283 ms	ca. 1,3	549 542 ms	437 559 ms	ca. 1,3	?	?	

Legende: siehe Seite 106

Tabelle 6.7: Messwerte der Tests für die Heuristik 6.4

Kapitel 7

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, grundlegende Konzepte für die auf Umformungsregeln basierende Optimierung der Ausführung von SPARQL-Anfragen zu entwickeln, diese Konzepte zu implementieren und Beispielheuristiken experimentell zu untersuchen. Hierzu wurden in Kapitel 2 zunächst die zugrundeliegenden Technologien RDF und SPARQL und die wichtigsten in diesem Zusammenhang auftretenden Begriffe vorgestellt. Weiterhin wurde hier ein erster Überblick über die Bearbeitung von Anfragen und die Einordnung möglicher Anfrageoptimierungen gegeben und Publikationen vorgestellt, welche sich mit verwandten Themen beschäftigen.

Als essentielle Grundlage wurde zunächst das SPARQL Query Graph Model (SQGM) spezifiziert (siehe Kapitel 3). Die Aufgabe dieses Modells ist die Repräsentation von SPARQL-Anfragen während des Optimierungsprozesses, welcher in dieser Arbeit entwickelt werden sollte. Demzufolge beziehen sich alle im Weiteren entwickelten Konzepte auf dieses Modell. Um die Entwicklung dieser Konzepte zu motivieren und sie zugleich in die Bearbeitung von SPARQL-Anfragen einordnen zu können, wurde ein möglicher Einsatz des SQGM in allen Schritten der Anfragebearbeitung diskutiert. Neben der formalen Definition des SQGM wurde eine Möglichkeit zur grafischen Darstellung dieser Modelle eingeführt.

Die in dieser Arbeit diskutierte Variante der Anfrageoptimierung basiert auf der Umformung des SQGM, welches die zu optimierende Anfrage repräsentiert. Diese Umformungen werden durch Regeln beschrieben. In der Arbeit wurden zwei Arten von Regeln eingeführt. Die Transformationsregeln, welche eine lokale Bedeutung haben und jeweils einen bestimmten Operator und dessen Nachbarn betrachten und verändern, und die komplexeren Restrukturierungsregeln, welche sich auf ein gesamtes SQGM beziehen. Während die Restrukturierungsregeln einerseits auf den Transformationsregeln basieren, bilden sie andererseits die Grundlage für die Heuristiken, welche für die diskutierte Anfrageoptimierung angewandt werden. Diese Beziehungen rechtfertigen den Begriff der regelbasierten Optimierung.

Um eine entsprechende Optimierung unter Verwendung des SQGM durchführen zu können, muss zunächst eine SQGM-Repräsentation der zu optimierenden Anfrage existieren. Für den dafür notwendigen Prozess der Erzeugung eines speziellen SQGM wurde im vierten Kapitel ein Vorgehen formal beschrieben. Dabei wird für alle Elemente von SPARQL-Anfragen eine Möglichkeit zu ihrer Modellierung angegeben. Nach der Erstellung eines SQGM muss dieses in eine für die regelbasierte Optimierung verwendbare Form gebracht werden. Das Kapitel 4 beinhaltet deswegen noch die Definition einer entsprechenden Normalform und die Spezifikation mehrerer Transformationsregeln, welche für eine entsprechende Umformung eingesetzt werden können. Die definierte Normalform legt bestimmte Eigenschaften fest, welche ein SQGM für die korrekte Ausführung der folgenden Optimierung besitzen muss. Abgeschlossen wird das vierte Kapitel mit der Vorstellung einer Reihe von weiteren Transformationsregeln zur Vereinfachung eines normalisierten SQGM. Hierbei werden Redundanzen entfernt und mögliche Widersprüche behandelt.

Neben der Entwicklung der theoretischen Konzepte bestand die Aufgabe dieser Arbeit in der praktischen Umsetzung dieser Konzepte und der experimentellen Untersuchung einiger Beispielheuristiken. Mit diesen Punkten befassen sich die Kapitel 5 und 6. Im fünften Kapitel werden

zunächst die entworfenen Modelle zur Umsetzung des SQGM und seiner Bestandteile vorgestellt und kurz auf deren Implementierung eingegangen. Unter Verwendung dieser Repräsentation wurde dann ein Regelausführungssystem entwickelt, welches ein SQGM mit Hilfe von Transformations- und Restrukturierungsregeln umformen kann. Die in der Arbeit spezifizierten Regeln wurden in diesem Zusammenhang ebenfalls implementiert. Für den Einsatz der praktisch umgesetzten Konzepte im Rahmen von tatsächlichen Anfragebearbeitungen, wurde eine prototypische Erweiterung der Anfragebearbeitungskomponente von ARQ entwickelt und die implementierten Datenstrukturen und Algorithmen integriert. Die Beschreibung dieser Integration ist ein weiterer Bestandteil des fünften Kapitels.

Die fundamentale Idee der regelbasierten Anfrageoptimierung besteht in der Anwendung von heuristischen Regeln, welche versprechen die Effizienz der Anfrageausführung zu erhöhen und dazu bestimmte Umformungen der Anfragen veranlassen. Im Rahmen dieser Arbeit wurden vier entsprechende Heuristiken beispielhaft entwickelt und ausführlich untersucht. Im Kapitel 6 wird für jede dieser vier Heuristiken zunächst die zugrundeliegende Vermutung erläutert. Daraufhin werden jeweils benötigte Transformations- und Restrukturierungsregeln spezifiziert. Diese wurden, in Hinblick auf eine experimentelle Untersuchung der Heuristiken, implementiert. Somit konnten unter Verwendung der in Kapitel 5 beschriebenen Implementierung verschiedene Messreihen für jede Heuristik durchgeführt werden. Die dabei eingesetzten Testanfragen wurden in Kapitel 6 vorgestellt. Anschließend werden jeweils die ermittelten Messwerte aufgelistet und diskutiert.

Die Auswertung der experimentellen Untersuchungen hat die folgenden Erkenntnisse geliefert. Zunächst ist festzustellen, dass die in dieser Arbeit entwickelten Konzepte im Rahmen der Bearbeitung von SPARQL-Anfragen eingesetzt werden können. Die Zeiten für ihre Anwendung sind im Vergleich zu den gemessenen Ausführungszeiten sehr gering. Das bedeutet, der Anteil für die regelbasierte Optimierung an der gesamten Zeit für die Bearbeitung einer Anfrage ist vernachlässigbar. Damit ist eine wichtige Voraussetzung für den Einsatz dieses Optimierungsschrittes erfüllt. Weiterhin wurden mit Hilfe der Messwerte die Vermutungen, welche den Heuristiken zugrundeliegen, bis auf wenige Ausnahmen bestätigt. In den überwiegenden Fällen konnten wesentliche Verkürzungen der Anfragebearbeitungszeiten festgestellt werden. Das rechtfertigt es, alle in dieser Arbeit formulierten Heuristiken in den entsprechenden Anwendungsszenarien einzusetzen.

Zusammenfassend kann festgestellt werden, dass die regelbasierte Optimierung während der Bearbeitung von SPARQL-Anfragen ein enormes Optimierungspotential birgt, wie die experimentellen Untersuchungen der Beispielheuristiken belegen. Weiterhin zeigen diese Experimente, dass die in der Arbeit entwickelten Konzepte und Techniken sich als Grundlagen für die regelbasierte Optimierung bewährt haben. Einen wesentlichen Anteil daran hat das hierfür entworfene SQGM als zugrundeliegende Repräsentationsform der SPARQL-Anfragen. Der folgende Ausblick zeigt Anknüpfungspunkte und skizziert mögliche Weiterentwicklungen.

Während das SQGM in dieser Arbeit für die regelbasierte Anfrageoptimierung eingesetzt wurde, ist es auch im Rahmen der anderen Schritte der Anfragebearbeitung einsetzbar. Bei der Entwicklung des SQGM wurde auf eine mögliche Erweiterbarkeit des Modells für andere Aufgabenbereiche geachtet. Als Hauptelemente eines SQGM wurden Operatoren verschiedenen Typs und Datenflüsse spezifiziert. Diese Elemente haben generelle und spezielle Eigenschaften. Im Hinblick auf einen umfassenderen Einsatz des SQGM könnten die Elemente um zusätzliche Eigenschaften erweitert werden. Ausserdem ist es beispielsweise möglich, neue Operatortypen zu spezifizieren und in das Modell aufzunehmen. Die objekt-orientierte Umsetzung erlaubt die problemlose Implementierung dieser Erweiterungen.

Ein möglicher Aufgabenbereich ist die bereits in Abschnitt 3.3.3 (ab Seite 36) skizzierte Einsetzung des SQGM im Rahmen der physischen Optimierung. Hierfür müssten die Grundlagen und theoretischen Konzepte geschaffen werden, um aus dem SQGM Ausführungspläne zu erzeugen. Entsprechende Grundlagen wären die formelle Erweiterung des SQGM und eine Anpassung der Implementierung. So könnten die Operatoren um Eigenschaften erweitert werden, welche die Algorithmen kennzeichnen, die zur Ausführung der entsprechenden Operationen vorgesehen werden. Weiterhin wäre eine zusätzliche Festlegung zu verwendender Indexstrukturen denkbar. Im Hinblick auf die kostenbasierte Auswahl könnten außerdem Eigenschaften für die Größe der abgefragten

Datenmengen, die Kosten der ausgewählten Algorithmen und die Größe der Zwischenlösungen eingeführt werden.

Als theoretische Konzepte für die physische Optimierung kommen beispielsweise Algorithmen zur Restrukturierung von Verbund- oder Vereinigungs-Bäumen, welche sich innerhalb eines SQGM befinden, in Frage. Diese Algorithmen hätten das Ziel, die Verbund- bzw. Vereinigungs-Operatoren in eine für die Ausführung möglichst effiziente Reihenfolge zu bringen. Für die Umformungen könnten sie Transformationsregeln wie die in Abschnitt 6.3.3 (ab Seite 97) spezifizierten verwenden. Die Entscheidung für bestimmte Reihenfolgen würde auf den oben erwähnten zusätzlichen Annotationen im SQGM und Kenntnissen über die abgefragten RDF-Datenmengen beruhen. Als solche Informationen über die Datenmengen kommen beispielsweise Schemadaten oder Statistiken in Frage.

Neben verschiedenster vorstellbarer Konzepte für die Erzeugung unterschiedlicher Ausführungspläne werden im Rahmen der physischen Optimierung mit kostenbasierter Auswahl noch Techniken für die Bestimmung des effizientesten Plans benötigt. Hierbei handelt es sich hauptsächlich um zu entwickelnde Kostenfunktionen und um Möglichkeiten für deren Evaluierung. Außerdem könnten Verfahren zur dynamischen Einschränkung des Suchraums oder Näherungsverfahren entworfen und implementiert werden.

Um das SQGM auch im Rahmen der Anfrageausführung einsetzen zu können, müssten die für die Operatoren auszuführenden Operationen implementiert werden. Für diese Implementierung kommen bei einigen Operationen sicherlich mehrere verschiedene Algorithmen oder Techniken in Frage. Es ist davon auszugehen, dass diese Umsetzungen bestimmte Eigenschaften haben. Das könnten bestimmte Anwendbarkeitsvoraussetzung, Statistiken wie durchschnittliche Ausführungskosten oder Angaben über Kombinationsmöglichkeiten sein. Demzufolge müssten neben der bloßen Umsetzung der Operationen auch Konzepte zur Bereitstellung dieser Informationen entwickelt werden. Hier zeigt sich ein direkter Zusammenhang zwischen der Anfrageausführung und der vorher stattfindenden Planerzeugung und -auswahl, welche auf diese Informationen Zugriff haben müsste.

Letztendlich könnte dann eine vollständige auf dem SQGM basierende Komponente zur Bearbeitung von SPARQL-Anfragen entwickelt werden. Hier müssten die einzelnen Phasen und Schritte der Anfragebearbeitung verknüpft werden und womöglich Repräsentationsformen für die Ergebnisse der Anfragen entwickelt werden.

Neben den vorgeschlagenen zusätzlichen Anwendungsmöglichkeiten des SQGM seien an dieser Stelle noch einige Ideen mögliche Erweiterungen des Modells skizziert. In der hier vorgestellten Definition des Modells sind Anpassungen der Lösungen nur mit Hilfe eines Lösungsanpassungs-Operators (siehe Definition 3.16 auf Seite 26) vorgesehen. Es wäre eine abgewandelte Form des SQGM denkbar, welche, ähnlich wie im Original-QGM [HFLP89], einige der Anpassungsaufgaben bereits den anderen variablen-bereitstellenden Operatoren überträgt. Vorstellbar wäre hier das Entfernen von Duplikaten bereits bei der Erzeugung von Zwischenlösungen, falls eine disjunkte Ergebnismenge gefordert wird, oder eine Sortierung der Zwischenlösungen. Die Eigenschaften der Operatoren müssten um Wahrheitswerte erweitert werden, welche eine Zusicherung dieser Anpassungen signalisieren. Durch entsprechende Maßnahmen könnten Anfragen unter Umständen effizienter ausgeführt werden.

Eine andere mögliche Erweiterung des SQGM ist die Zuordnung von Wertebeschränkungen zu Verbund-Operationen und nicht nur zu Graphmuster-Operatoren. Dadurch werden Verbund-Operationen ähnlich dem *bedingten Verbund* (engl.: theta-join) der relationalen Algebra möglich. In diesem Rahmen müsste untersucht werden, ob diese Möglickeit für SPARQL-Anfragen sinnvoll ist und sich auf die Effizienz der Anfrageausführungen auswirkt.

Das letzte hier skizzierte Einsatzszenario des SQGM, sind Untersuchungen möglicher SPARQL-Erweiterungen und damit verbundener Optimierungsansätze. So ist es denkbar, SPARQL um Unteranfragen oder Sichten zu erweitern. Für deren Integration bietet sich das SQGM wegen seiner Flexibilität und Erweiterbarkeit an. Die Unteranfragen und die durch ihre jeweilige Definition substituierten Sichten könnten durch SQGMs repräsentiert werden, welche in das Modell der Hauptanfrage eingebettet sind. Auf diese Art könnten sie in den Optimierungsprozess der gesamten Anfrage einbezogen werden.

Neben den vier in dieser Arbeit formulierten Heuristiken für die regelbasierte Optimierung sind viele andere denkbar. Die im Rahmen der Arbeit entworfenen Konzepte, deren Implementierung und die entwickelte Testumgebung bieten alle Grundlagen zu einer ausführlichen Untersuchung weiterer Heuristiken. Bei deren Entwicklung kann man sich an dem für die vier Beispielheuristiken angewandten Vorgehen orientieren. Zum Abschluss des Ausblicks werden im Folgenden einige Anregungen für weitere mögliche Heuristiken skizziert.

Die vier Beispielheuristiken setzen alle dasselbe Anwendungsszenario voraus. Sie wurden für Anfragen formuliert, welche durch die ARQ-Anfragebearbeitungskomponente auf persistent gespeicherten RDF-Datenmengen ausgeführt werden. Dabei profitieren sie vom Fast-Path-Algorithmus (siehe Abschnitt 6.3.1 ab Seite 92). Es sind sicherlich noch weitere Möglichkeiten zur Ausnutzung dieser Besonderheit des Anwendungsszenarios denkbar. Beispielsweise könnten verschiedene RDF-Graph-Klauseln (siehe Abschnitt 2.3.7 ab Seite 12), welche sich auf dieselben Datenquellen beziehen, zusammengefasst werden.

Neben dem hier vorausgesetzten Anwendungsszenario könnten Heuristiken für viele andere Situationen spezifiziert werden. Womöglich kommen die in diesem Zusammenhang aufgestellten Vermutungen zu dem Schluss, dass dieselben Restrukturierungsregeln anzuwenden sind, wie bei den in dieser Arbeit vorgestellten Beispielheuristiken. Es könnte natürlich auch möglich sein, dass neue Restrukturierungsregeln mit neuen Transformationsregeln spezifiziert werden müssen. Diese Restrukturierungsregeln könnten in ihren Zielen den hier spezifizierten widersprechen. Während beispielsweise die Regel PullAllJoinedUnions (siehe Definition 6.2 auf Seite 117) versucht, sämtliche Vereinigungs-Operatoren in Richtung Wurzel des SQGM zu propagieren, könnte es in anderen Situationen sinnvoll sein, die Verbund-Operatoren in Wurzel-Richtung zu propagieren, wodurch die Vereinigungs-Operatoren in die entgegengesetzte Richtung verschoben werden.

Auch wenn das ARQ zugrundeliegende Semantic-Web-Framework Jena von der tatsächlichen Form, in der die abgefragten RDF-Graphen vorliegen, abstrahiert (siehe Abschnitt 5.1 ab Seite 71), kann diese Form einen Einfluss auf die einzusetzenden Heuristiken haben. Neben der in dieser Arbeit betrachteten Form der Speicherung in Datenbanken, könnten andere Heuristiken formuliert werden, welche sich auf Tripelmengen im Hauptspeicher oder durch Inferenz gefolgerte virtuelle RDF-Graphen beziehen. Hierfür müssten die Eigenheiten bei der Anfrageausführung für diese Formen untersucht und ausgenutzt werden.

Neben ARQ existieren weitere Systeme, welche den Zugriff auf RDF-Datenmengen mittels SPARQL-Anfragen erlauben und entsprechende Anfragebearbeitungskomponenten beinhalten. Als Beispiele seien hier Sesame[1] [BKvH02] und die SPARQL-Komponente sparql-p [Her05] der Bibliothek RDFLib[2] genannt. In diesen sind unter Umständen andere Konzepte zur Ausführung der Anfragen implementiert. Demzufolge könnte man auch für diese Systeme Heuristiken für die regelbasierte Optimierung auf Basis des SQGM formulieren und untersuchen. Entsprechende experimentelle Untersuchungen erfordern jedoch, dass die entwickelten Umsetzungen der hier entworfenen Konzepte in die Anfragebearbeitung dieser Systeme integriert werden. Eine solche Integration könnte sich an der für diese Arbeit entwickelten Einbettung in ARQ (siehe Abschnitt 5.3 ab Seite 75) orientieren.

Den vier Beispielheuristiken aus Kapitel 6 ist gemein, dass die von ihnen angegebenen Restrukturierungsregeln immer angewandt werden, wenn die Voraussetzungen der Heuristiken erfüllt sind. Es sind jedoch auch Heuristiken vorstellbar, die sich unter verschiedenen Bedingungen für die Anwendung unterschiedlicher Restrukturierungsregeln entscheiden. Eine andere Klasse von möglichen Heuristiken könnte, wie in Abschnitt 6.3.6 (Seite 108) angedeutet, Schemadaten zu den abgefragten Datenmengen einsetzen. Dadurch können flexiblere Entscheidungen zu Gunsten bestimmter Regeln getroffen werden und Ausnahmen wie bei den Untersuchungen der Heuristiken 6.1 (Abschnitt 6.3.5 ab Seite 104) und 6.2 (Abschnitt 6.4.1 ab Seite 109) vermieden bzw. gesondert behandelt werden.

[1] `http://www.openrdf.org/`
[2] `http://rdflib.net/`

Anhang A

Testanfragen

Die folgenden Seiten zeigen die SPARQL-Anfragen, welche zur Untersuchung der Heuristiken in Kapitel 6 verwendet wurden. Die folgenden Anfragen stellen jeweils nur eine verkürzte Variante der tatsächlich ausgeführten Anfragen dar. Um sie platzsparend und übersichtlich darstellen zu können, wurde jeweils der Prolog mit den PREFIX-Klauseln für Präfix-Spezifikationen zur Festlegung von URI-Namensräumen weggelassen. Dieser Kopf ist bei allen Anfragen derselbe und lautet:

```
PREFIX ub:   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

Anfrage A.1

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:name ?n . }
}
```

Anfrage A.2

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:takesCourse ?c . }
}
```

Anfrage A.3

```
SELECT
 ?n ?c
WHERE
{
   { ?s ub:takesCourse ?c . }
   { ?s ub:name ?n . }
}
```

Anfrage A.4

```
SELECT
 ?s ?c
WHERE
{
   { ?s ub:name ?n . }
   { ?s ub:name ?n . }
}
```

Anfrage A.5

```
SELECT
 ?n ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:takesCourse ?c . }
   { ?s ub:name ?n . }
}
```

Anfrage A.6

```
SELECT
 *
WHERE
{
   { ?t ub:teacherOf ?c . }
   { ?t ub:emailAddress ?e . }
   { ?s ub:takesCourse ?c . }
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:name ?n . }
}
```

Anfrage A.7

```
SELECT
 ?s ?n ?c
WHERE
{
   {
      ?s rdf:type ub:GraduateStudent .
      OPTIONAL { ?s ub:name ?n . }
   }
   { ?s ub:takesCourse ?c . }
}
```

Anfrage A.8

```
SELECT
 ?s ?n ?c
WHERE
{
   {
      ?s rdf:type ub:GraduateStudent .
      OPTIONAL { ?s ub:takesCourse ?c . }
   }
   { ?s ub:name ?n . }
}
```

Anfrage A.9

```
SELECT
 *
WHERE
{
   {
      ?t ub:teacherOf ?c .
      OPTIONAL { ?t ub:emailAddress ?e . }
   }

   ?s ub:takesCourse ?c .

   {
      ?s rdf:type ub:GraduateStudent .
      OPTIONAL { ?s ub:name ?n . }
   }
}
```

Anfrage A.10

```
SELECT
 *
WHERE
{
   { ?s ?t ub:GraduateStudent . }
   { ?s ub:name ?n . }
}
```

Anfrage A.11

```
SELECT
 *
WHERE
{
   { ?s ?t ub:GraduateStudent . }
   { ?s ub:takesCourse ?c . }
}
```

Anfrage A.12

```
SELECT
 *
WHERE
{
   {
      ?s ?t ub:GraduateStudent .
      OPTIONAL { ?s ub:name ?n . }
   }
   { ?s ub:takesCourse ?c . }
}
```

Anfrage A.13

```
SELECT
 *
WHERE
{
   {
      ?s ?t ub:GraduateStudent .
      OPTIONAL { ?s ub:takesCourse ?c . }
   }
   { ?s ub:name ?n . }
}
```

Anfrage A.14

```
SELECT
 *
WHERE
{
   {
      ?s rdf:type ub:GraduateStudent .
      OPTIONAL { ?s ub:name ?n . }
   }

   ?s ?tc ?c .

   {
      ?t ub:teacherOf ?c .
      OPTIONAL { ?t ub:emailAddress ?e . }
   }
}
```

Anfrage A.15

```
SELECT
 *
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:name ?n . }
   { ?s ?tc ?c . }
   { ?t ub:teacherOf ?c . }
   { ?t ub:emailAddress ?e . }
}
```

Anfrage A.16

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:name ?n .
     FILTER regex( ?n, "1" )
   }
}
```

Anfrage A.17

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:takesCourse ?c .
     FILTER regex( str(?c), "2" )
   }
}
```

Anfrage A.18

```
SELECT
 ?n ?c
WHERE
{
   { ?s ub:takesCourse ?c . }
   { ?s ub:name ?n . }

   FILTER regex( str(?c), "2" )
   FILTER regex( ?n, "1" )
}
```

Anfrage A.19

```
SELECT
 ?n ?c
WHERE
{
   {
      ?s ub:takesCourse ?c .
      FILTER regex( str(?c), "2" )
   }
   {
      ?s ub:name ?n .
      FILTER regex( ?n, "1" )
   }
}
```

Anfrage A.20

```
SELECT
 ?n ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   {
      ?s ub:takesCourse ?c .
      FILTER regex( str(?c), "2" )
   }
   {
      ?s ub:name ?n .
      FILTER regex( ?n, "1" )
   }
}
```

Anfrage A.21

```
SELECT
 ?n ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent . }
   { ?s ub:takesCourse ?c . }
   { ?s ub:name ?n . }

   FILTER regex( str(?c), "2" )
   FILTER regex( ?n, "1" )
}
```

Anfrage A.22

```
SELECT
 ?n ?c
WHERE
{
   {
      ?s rdf:type ub:GraduateStudent .
      ?s ub:takesCourse ?c .
      FILTER regex( str(?c), "2" )
   }
   {
      ?s ub:name ?n .
      FILTER regex( ?n, "1" )
   }
}
```

Anfrage A.23

```
SELECT
 ?n ?c
WHERE
{
   {
      ?s rdf:type ub:GraduateStudent .
      FILTER ( ! regex( str(?s), "2" ) )
   }

   {
      ?s ub:takesCourse ?c .
      FILTER regex( str(?s), "2" )
   }

   ?s ub:name ?n .
}
```

Anfrage A.24

```
SELECT
 ?s ?n ?c
WHERE
{
   ?s rdf:type ub:GraduateStudent .
   ?s ub:takesCourse ?c .
   OPTIONAL { ?s ub:name ?n . }

   FILTER regex( str(?c), "2" )
}
```

Anfrage A.25

```
SELECT
 ?s ?n ?c
WHERE
{
   ?s rdf:type ub:GraduateStudent .
   ?s ub:name ?n .
   OPTIONAL { ?s ub:takesCourse ?c . }

   FILTER regex( ?n, "1" )
}
```

Anfrage A.26

```
SELECT
 *
WHERE
{
   { ?s ?t ub:GraduateStudent . }
   {
      ?s ub:name ?n .
      FILTER regex( ?n, "1" )
   }
}
```

Anfrage A.27

```
SELECT
 *
WHERE
{
   { ?s ?t ub:GraduateStudent . }
   {
      ?s ub:takesCourse ?c .
      FILTER regex( str(?c), "2" )
   }
}
```

Anfrage A.28

```
SELECT
 *
WHERE
{
   ?s ?t ub:GraduateStudent .
   ?s ub:takesCourse ?c .
   OPTIONAL { ?s ub:name ?n . }

   FILTER regex( str(?c), "2" )
}
```

Anfrage A.29

```
SELECT
 *
WHERE
{
   ?s ?t ub:GraduateStudent .
   ?s ub:name ?n .
   OPTIONAL { ?s ub:takesCourse ?c . }

   FILTER regex( ?n, "1" )
}
```

Anfrage A.30

```
SELECT
 ?s
WHERE
{
   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }

   ?s ub:name ?n .
}
```

Anfrage A.31

```
SELECT
 ?s
WHERE
{
   ?s ub:name ?n .

   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }
}
```

Anfrage A.32

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }

   ?s ub:takesCourse ?c .
}
```

Anfrage A.33

```
SELECT
 ?s ?c
WHERE
{
   ?s ub:takesCourse ?c .

   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }
}
```

Anfrage A.34

```
SELECT
 ?s ?c
WHERE
{
   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }

   ?s ub:takesCourse ?c
   OPTIONAL { ?s ub:name ?n }
}
```

Anfrage A.35

```
SELECT
 ?s ?c
WHERE
{
   ?s ub:takesCourse ?c
   OPTIONAL { ?s ub:name ?n }

   { ?s rdf:type ub:GraduateStudent }
   UNION
   { ?s rdf:type ub:UndergraduateStudent }
}
```

Anfrage A.36

```
SELECT
 ?s
WHERE
{
   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }

   ?s ub:name ?n .
}
```

Anfrage A.37

```
SELECT
 ?s
WHERE
{
   ?s ub:name ?n .

   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }
}
```

Anfrage A.38

```
SELECT
 ?s ?c
WHERE
{
   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }

   ?s ub:takesCourse ?c .
}
```

Anfrage A.39

```
SELECT
 ?s ?c
WHERE
{
   ?s ub:takesCourse ?c .

   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }
}
```

Anfrage A.40

```
SELECT
 ?s ?c
WHERE
{
   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }

   ?s ub:takesCourse ?c
   OPTIONAL { ?s ub:name ?n }
}
```

Anfrage A.41

```
SELECT
 ?s ?c
WHERE
{
   ?s ub:takesCourse ?c
   OPTIONAL { ?s ub:name ?n }

   { ?s ?t ub:GraduateStudent }
   UNION
   { ?s ?t ub:UndergraduateStudent }
}
```

Abkürzungsverzeichnis

API	-	Application Programming Interface
DBMS	-	Database Management System
QEP	-	Query Evaluation Plan, Query Evaluation Program
QGM	-	Query Graph Model
RDBMS	-	Relational Database Management System
RDF	-	Resource Description Framework
SQGM	-	SPARQL Query Graph Model
URI	-	Uniform Resource Identifiers
W3C	-	World Wide Web Consortium
XML	-	Extensible Markup Language

Literaturverzeichnis

[ACK+01] Alexaki, Sofia, Vassilis Christophides, Grigoris Karvounarakis, Dimitris Plexousakis und Karsten Tolle: *The ICS-FORTH RDF Suite: Managing Voluminous RDF Description Bases.* In: *Proceedings of the 2nd International Workshop on the Semantic Web (SemWeb'01) in conjunction with 10th International World Wide Web Conference (WWW10)*, Seiten 1–13, Mai 2001. `http://www.ics.forth.gr/isl/publications/paperlink/semweb2001.pdf`.

[ACK+02] Alexaki, Sofia, Vassilis Christophides, Grigoris Karvounarakis, Dimitris Plexousakis und M. Schol: *RQL: A Declarative Query Language for RDF.* In: *Proceedings of the Eleventh International World Wide Web Conference (WWW'02)*, Mai 2002.

[Bec04a] Beckett, Dave: *RDF/XML Syntax Specification (Revised), W3C Recommendation.* W3C (World Wide Web Consortium), Februar 2004. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[Bec04b] Beckett, Dave: *Turtle - Terse RDF Triple Language*, 2004. `http://www.ilrt.bris.ac.uk/discovery/2004/01/turtle/`.

[Bec06] Beckett, Dave: *SPARQL Query Results XML Format, W3C Candidate Recommendation.* W3C (World Wide Web Consortium), April 2006. `http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/`.

[BG03] Beckett, Dave und Jan Grant: *Mapping Semantic Web Data with RDBMSes*, Januar 2003. `http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report`.

[BG04] Brickley, Dan und R.V. Guha: *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation.* W3C (World Wide Web Consortium), Februar 2004. `http://www.w3.org/TR/rdf-schema/`.

[BKvH02] Broekstra, Jeen, Arjohn Kampman und Frank van Harmelen: *Sesame: An Architecture for Storing and Querying RDF and RDF Schema.* In: *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, Band 2342 der Reihe *Lecture Notes in Computer Science*, Seiten 54–68. Springer, Juni 2002. `http://wwwis.win.tue.nl:8080/~jbroekst/papers/ISWC02.pdf`.

[BLFM98] Berners-Lee, T., R. Fielding und L. Masinter: *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax.* IETF (Internet Engineering Task Force), August 1998. `http://www.isi.edu/in-notes/rfc2396.txt`.

[BLHL01] Berners-Lee, Hendler und Lassila: *The Semantic Web.* Scientific American, Seiten 34–43, Mai 2001.

[Car02] Carroll, Jeremy J.: *Matching RDF graphs.* In: Horrocks, Ian und James Hendler (Herausgeber): *Proceedings of the 1st International Semantic Web Conference (ISWC)*, Seiten 5–15, 2002.

[CDD+03] Carroll, Jeremy J., Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne und Kevin Wilkinson: *Jena: Implementing the Semantic Web Recommendations.* HP Laboratories Bristol, Dezember 2003. http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf.

[CDES05] Chong, Eugene, Souripriya Das, George Eadon und Jagannathan Srinivasan: *An Efficient SQL-based RDF Querying Scheme.* In: *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Trondheim, Norway*, 2005.

[CGK05] Chen, L., A. Gupta und M. E. Kurul: *A Semantic-aware RDF Query Algebra.* In: *Proceedings of the 12th International Conference on Management of Data (COMAD)*, Dezember 2005. http://www.sdsc.edu/~gupta/publications/comad_chen.pdf.

[Cla05] Clark, Kendall Grant: *RDF Data Access Use Cases and Requirements, W3C Working Draft.* W3C (World Wide Web Consortium), März 2005. http://www.w3.org/TR/2005/WD-rdf-dawg-uc-20050325/.

[Cla06] Clark, Kendall Grant: *SPARQL Protocol for RDF, W3C Candidate Recommendation.* W3C (World Wide Web Consortium), April 2006. http://www.w3.org/TR/2006/CR-rdf-sparql-protocol-20060406/.

[Cyg05] Cyganiak, Richard: *A relational algebra for SPARQL.* HP Laboratories Bristol, September 2005. http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html.

[dBFT05] Bruijn, Jos de, Enrico Franconi und Sergio Tessaris: *Logical Reconstruction of normative RDF.* In: *Proceedings of the Workshop on OWL Experiences and Directions (OWLED 2005)*, November 2005.

[Dod06] Dodds, Leigh: *SPARQLing Services.* In: *Proceedings of the XTech Confererence 2006*, Mai 2006. http://xtech06.usefulinc.com/schedule/paper/61.

[FHVB04] Frasincar, Flavius, Geert-Jan Houben, Richard Vdovjak und Peter Barna: *RAL: An Algebra for Querying RDF.* Journal of World Wide Web, Internet and Web Information Systems (WWW), 7(1):83–109, 2004. http://wwwis.win.tue.nl/~rvdovjak/publications/RALwwwj2004.pdf.

[GHM04] Gutierrez, Claudio, Carlos Hurtado und Alberto O. Mendelzon: *Foundations of Semantic Web Databases.* In: *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, Seiten 95–106, Juni 2004.

[GMWU02] Garcia-Molina, Hector, Jennifer Widom und Jeffrey D. Ullman: *Database Systems: The Complete Book.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002. ISBN 0-13-031995-3.

[GPH05] Guo, Y., Z. Pan und J. Heflin: *LUBM: A Benchmark for OWL Knowledge Base Systems.* Journal of Web Semantics, 3(2):158–182, 2005. http://www.websemanticsjournal.org/ps/pub/2005-16.

[GS05] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler: *Datenbank: Implementierungstechniken.* mitp-Verlag/Bonn, Zweite Auflage, 2005. ISBN 3-8266-1438-0.

[Har05] Harris, Stephen: *SPARQL query processing with conventional relational database systems.* In: *Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base System (SSWS 2005)*, 2005. http://eprints.ecs.soton.ac.uk/11126/.

[Hay04] HAYES, PATRICK: *RDF Semantics, W3C Recommendation.* W3C (World Wide Web Consortium), Februar 2004. http://www.w3.org/TR/rdf-mt/.

[HBEV04] HAASE, PETER, JEEN BROEKSTRA, ANDREAS EBERHART und RAPHAEL VOLZ: *A comparison of RDF query languages.* In: *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, Band 3298 der Reihe *Lecture Notes in Computer Science.* Springer, November 2004. http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf.

[HCL+90] HAAS, LAURA M., WALTER CHANG, GUY M. LOHMAN, JOHN MCPHERSON, PAUL F. WILMS, GEORGE LAPIS, BRUCE G. LINDSAY, HAMID PIRAHESH, MICHAEL J. CAREY und EUGENE J. SHEKITA: *Starburst Mid-Flight: As the Dust Clears.* IEEE Transactions on Knowledge and Data Engineering, 2(1):143–160, März 1990.

[HD05] HARTH, ANDREAS und STEFAN DECKER: *Optimized Index Structures for Querying RDF from the Web.* In: *Proceedings of the 3rd Latin American Web Congress*, Seiten 71–80, Oktober 2005.

[Her05] HERMAN, IVAN: *SPARQL in RDFLib (Version 2.1)*, Oktober 2005. http://dev.w3.org/cvsweb/~checkout~/2004/PythonLib-IH/Doc/sparqlDesc.ht%ml.

[HFLP89] HAAS, LAURA M., JOHANN CHRISTOPH FREYTAG, GUY M. LOHMAN und HAMID PIRAHESH: *Extensible Query Processing in Starburst.* In: CLIFFORD, JAMES, BRUCE G. LINDSAY und DAVID MAIER (Herausgeber): *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Seiten 377–388. ACM Press, Mai 1989.

[HG03] HARRIS, STEPHEN und NICHOLAS GIBBINS: *3store: Efficient Bulk RDF Storage.* In: *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, Oktober 2003. http://km.aifb.uni-karlsruhe.de/ws/psss03/proceedings/harris-et-al.pdf.

[KC04] KLYNE, GRAHAM und JEREMY J. CARROLL: *Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation.* W3C (World Wide Web Consortium), Februar 2004. http://www.w3.org/TR/rdf-concepts/.

[KM01] KOIVUNEN und MILLER: *W3C Semantic Web Activity.* In: *Proceedings of the Semantic Web Kick-off Seminar.* W3C (World Wide Web Consortium), November 2001. http://www.w3.org/2001/12/semweb-fin/w3csw.

[Mey01] MEYERS LEXIKONREDAKTION (Herausgeber): *Duden Informatik – Ein Fachlexikon für Studium und Praxis.* Dudenverlag, Dritte Auflage, 2001. ISBN 3-411-05233-3.

[MKC+02] MAGANARAKI, AIMILIA, GRIGORIS KARVOUNARAKIS, VASSILIS CHRISTOPHIDES, DIMITRIS PLEXOUSAKIS und TA TUAN ANH: *Ontology Storage and Querying.* Foundation for Research and Technology Hellas, Institute of Computer Science, April 2002. Technical Report No 308 http://www.ics.forth.gr/isl/publications/paperlink/tr308.pdf.

[MM04] MANOLA, FRANK und ERIC MILLER: *RDF Primer, W3C Recommendation.* W3C (World Wide Web Consortium), Februar 2004. http://www.w3.org/TR/rdf-primer/.

[PAG06] PEREZ, JORGE, MARCELO ARENAS und CLAUDIO GUTIERREZ: *Semantics and Complexity of SPARQL*, Mai 2006. http://www.dcc.uchile.cl/~cgutierr/ftp/sparql-ext.pdf.

[PH04] PAN, ZHENGXIANG und JEFF HEFLIN: *DLDB: Extending Relational Databases to Support Semantic Web Queries.* Deptartment of Computer Science and Engineering, Lehigh University, 2004. Technical Report LU-CSE-04-006 `www3.lehigh.edu/images/userImages/jgs2/Page_3813/LU-CSE-04-006.pdf`.

[PS06] PRUD'HOMMEAUX, ERIC und ANDY SEABORNE: *SPARQL Query Language for RDF, W3C Candidate Recommendation.* W3C (World Wide Web Consortium), April 2006. `http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/`.

[RvGB03] REGGIORI, ALBERTO, DIRK-WILLEM VAN GULIK und ZAVISA BJELOGRLIC: *Indexing and retrieving Semantic Web resources: the RDFStore model.* In: *Proceedings of the SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, November 2003. `http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/asem%antics-2.html`.

[S06] *The SeRQL query language.* In *User Guide for Sesame*, 2006. `http://www.openrdf.org/doc/sesame/users/ch06.html`.

[Sch98] SCHNEIDER, HANS-JOCHEN (Herausgeber): *Lexikon Informatik und Datenverarbeitung.* R. Oldenbourg Verlag, Vierte Auflage, 1998. ISBN 3-486-24538-4.

[Sch04] SCHNEIDER, MARKUS: *Implementierungskonzepte für Datenbanksysteme.* Springer-Verlag, 2004. ISBN 3-540-41962-4.

[Sea06] SEABORNE, ANDY: *Implementing SPARQL.* In: *Proceedings of the 2006 Jena User Conference*, Mai 2006. `http://jena.hpl.hp.com/juc2006/proceedings/seaborne/slides.ppt`.

[SKS97] SILBERSCHATZ, ABRAHAM, HENRY F. KORTH und S. SUDERSHAN: *Database System Concepts.* McGraw-Hill, Inc., Third Auflage, 1997. ISBN 0-07-114810-8.

[SKT05] SERFIOTIS, GEORGE, IOANNA KOFFINA und VAL TANNEN: *Containment and Minimization of RDF/S Query Patterns.* In: *Proceedings of the 4th Hellenic Data Management Symposium (HDMS'05)*, August 2005. `http://www.ics.forth.gr/isl/publications/paperlink/hdms_containment.pdf%`.

[SVBH04] STUCKENSCHMIDT, HEINER, RICHARD VDOVJAK, JEEN BROEKSTRA und GEERT-JAN HOUBEN: *Towards Distributed Processing of RDF Path Queries.* International Journal on Web Engineering and Technology, Februar 2004. `http://wwwis.win.tue.nl:8080/~jbroekst/papers/IJWET.pdf`.

[TCK05] THEOHARIS, YANNIS, VASSILIS CHRISTOPHIDES und GRIGORIS KARVOUNARAKIS: *Benchmarking Database Representations of RDF/S Stores.* In: *Proceedings of the Fourth International Semantic Web Conference (ISWC2005)*, November 2005. `http://139.91.183.30:9090/RDF/publications/RBenchFinal.pdf`.

[WSKR03] WILKINSON, KEVIN, CRAIG SAYERS, HARUMI KUNO und DAVE REYNOLDS: *Efficient RDF storage and retrieval in Jena2.* In: *Proceedings of the 1st VLDB Workshop on Semantic Web and Databases*, Seiten 131–150, September 2003.

[YM98] YU, CLEMENT T. und WEIYI MENG: *Principles of Database Query Processing for advanced Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-434-0.