Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZAI-BSC-2014-10

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Auswertung von SPARQL-Anfragen mit relationaler Speicherung

Lars Runge, Sebastian Schrage

am Lehrstuhl für

Fakultät für Mathematik und Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

2. April 2014

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 2. April 2014

Bachelorarbeit

Auswertung von SPARQL-Anfragen mit relationaler Speicherung

Lars Runge, Sebastian Schrage

2. April 2014

Betreut durch Prof. Wolfgang May
Arbeitsgruppe Datenbanken
Georg-August-Universität Göttingen

Inhaltsverzeichnis

Abbildung 0.1.: SQL Datenbank mit der Funktionalität eines RDF Models
Quelle: http://static.fjcdn.com/pictures/Today_45a56e_997902.jpg

1. Einführung

1.1. Motivation

Als Datenbanken nur den Anspruch erfüllen mussten Daten zu speichern und effektiv ausgeben zu können, war das relationale Datenmodell über lange Zeit in vieler Hinsicht ausreichend. Doch gerade mit Blick auf die Möglichkeiten des Internets, wurden neue Anforderungen an eine moderne Datenbank gestellt, die zu der Entwicklung neuer Datenmodelle führten. Dabei sollen Datenbanken vorallem flexibel sein und den Menschen auch bei logisch auswertbaren Problem unterstützen.

In einer geschlossenen Welt, in der davon ausgegangen wird, dass das gesammelte Wissen vollständig ist, würden klassische Datenbanken eine hoch optimierte und stabile Umgebung bieten. Doch wenn man die Möglichkeit unbekannten Wissens in Betracht zieht, wie es zum Beispiel in der Internet-Umgebung vorkommen kann, so stehen nahezu unerschöpfliche Mengen an Informationen zu Verfügung. Diese müssen interpretiert und einer semantischen Bedeutung zugeordnet werden, da sie sonst wertlos sind. Vorallem wird aber die Möglichkeit von Wissenslücken in das System eingeführt. Bei diesen Mengen an Daten kann ein Mensch diese Aufgabe nicht mehr effektiv bewältigen und das relationale Datenmodell ist für eine solche Betrachtungsweise nicht ausgelegt. Hier kommt das sogenannte SemanticWeb ins Spiel und mit ihm das Resource Description Framework (RDF) (s. auf Seite 13). In diesem Datenmodell können Ontologien mit der Web Ontology Language (OWL) definiert werden, die „(manchmal) ein umfassendes Wissen über das Konzeptlevel einer Anwendung“[10] bieten soll. Dazu gehört die Charakterisierung der relevanten Resourcenklassen, die Klassenhierarchie, sowie die Eigenschaften und Beziehungen zwischen diesen.

Doch hat dieser Gewinn an Informationen auch seinen Preis: verglichen mit relationalen Datenschemas, wie SQL (s. auf Seite 9), ist RDF langsamer. Aus diesem Umstand ist die Idee geboren, ein RDF-Modell in eine SQL-Datenbank einzubetten und dabei die RDF - Funktionalitäten beizubehalten.

Aufgabe dieser Arbeit ist es daher, ein semistrukturiertes Datenschema aus RDF in das voll strukturierte Datenmodell einer relationalen Speicherung umzuwandeln. Über diese können dann für SPARQL Anfragen äquivalente SQL Anfragen konstruiert werden.

Das Ziel dieser Umwandlung ist es für RDF Datensätze, die sich wenig ändern, aber viel benutzt werden, einen schnellen Zugriff ohne größere Einschränkungen der Funktionalität zu gewähren. Damit werden die Stärken von SQL und RDF gewissermaßen vereint und helfen sich gegenseitig, ihre Schwächen zu überwinden. Die Effizienz solcher Verfahren im Vergleich zur nativen RDF Speicherung wurde bereits durch den Berliner SPARQL Benchmark nachgewiesen. [2]

Im Rahmen dieser Arbeit wurden zwei Programme geschrieben: Zum einen das Programm „RDF2SQL”, das ein RDF Modell in eine relationale SQL Datenbank umwandelt (siehe [10],[14] & auf Seite 20). Zum Anderen das Programm „Query Converter”, dieses

konvertiert SPARQL Anfragen in äquivalente SQL-Anfragen (s. auf Seite 28).

1.2. Methoden

1.2.1. Tools

Im Rahmen dieser Arbeit wurden für die SQL-Anfragen Ojdbc 6 auf eine Oracle SQL Datenbank der Version 11.2.0.3.0 verwendet. Außerdem wurde für die SPARQL-Anfragen die Jena-Version 2.9.4 benutzt.

1.2.2. Datensatz - Mondial

Diese Programme wurden mit Hilfe der *MONDIAL*-Datenbank als Anwendungsbeispiel entwickelt. Die *MONDIAL*-Datenbank beinhaltet geographische Daten, die mit einer Reihe von Web Datenquellen im Jahre 1998 zusammengestellt wurden**(author?)** [12]. Zu diesen gehören zum Beispiel das CIA World Factbook [1], ein Ausschnitt der Karlsruher TERRA Datenbank und der International Atlas von Kümmerly & Frey, Rand McNally und Westermann. Entwickelt wurde die *MONDIAL*-Datenbank als eine Anwendungsstudie für das F-Logic System FLORID.

Gespeichert werden verschiedene Informationen zu Städten, Ländern, Organisationen sowie geographischen Landmarken, wie Flüsse und Berge. Die gesammelten Daten sind in den Formaten F-Logic, SQL, XML, RDF/OWL und Datalog unter [12] zu finden. Das im Praktikumsbericht vorgestellte Programm *RDF2SQL* (s. auf Seite 20) wertet die Meta-Informationen der RDF-Datenbank aus und erstellt ein passendes ER-Modell für eine SQL-Datenbank. Die generierte SQL-Datenbank unterscheidet sich dabei geringfügig von der originalen SQL-Datenbank von *MONDIAL*. Zudem müssen Meta-Tabellen angelegt werden, auf die der *Query Converter* (s. auf Seite 28) aufbaut.

2. Grundlagen

In diesem Kapitel werden grundlegende Kenntnisse über den allgemeinen Aufbau von RDF und SQL Datenbanken sowie deren standardisierte Anfragesprachen SPARQL und SQL vermittelt. Außerdem wird auf verschiedene relevante Feinheiten eingegangen, die diese beiden Modelle unterscheiden und die Konvertierung erschweren.

2.1. Syntaxbäume

Syntaxbäume sind baumartige Abbildungen, um die Ableitung eines Wortes einer Grammatik besser darstellen zu können. Für jede kontextfreie Sprache, und damit auch jede Anfragesprache, lassen sich Syntaxbäume konstruieren. Die Struktur des Baumes ist dabei geordnet, d.h. die Position jedes Kindes eines Knotens ist festgelegt. Dies ist entscheidend, da bei manchen Knoten die Bedeutung der Kinder abhängig von ihrer Position ist.

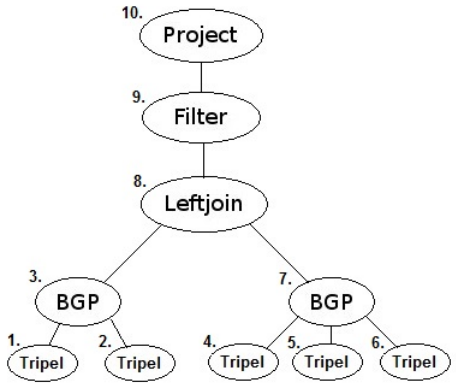

Abbildung 2.1.: Syntaxbaum einer SPARQL-Anfrage mit post-order Reihenfolge

Eine konkrete Anfrage an eine Datenbank entspricht einer Ableitung der zugehörigen Anfragesprache und kann daher mit einem Syntaxbaum dargestellt werden. Durch den Syntaxbaum und eine Traversierungsmethode steht damit auch die Reihenfolge fest, in der die Knoten des Baumes zur Auswertung betrachtet werden müssen. Die Auswertung von Anfragen ist in einer iterativen Reihenfolge definiert, weshalb diese daher auch für den Query Converter (s. auf Seite 28) verwendet wird. Abbildung 2.1 zeigt den Syntaxbaum einer beispielhaften SPARQL-Anfrage und die Evaluierungsreihenfolge ihrer Knoten. Jeder Knoten steht für eine bestimmte Operation oder einen Operator und jede Kante verweist auf die Operanden. In unserem Fall gibt es null-, ein-, zwei- und n-stellige Knoten. Nullstellige Knoten sind zwangsläufig die Blätter des Baumes. In einem SPARQL Syntaxbaum sind dies die Tripel. Zweistellige Knoten sind Konjugationen und Joins auf dessen Kindern, während n-stellige Knoten Basic Graph Patterns (BGP) darstellen. Alle anderen Operationen kommen mit einer Kante aus.

2.2. Strukturierte und semistrukturierte Daten

2.2.1. Strukturierte Daten

Unter strukturierten Daten versteht man ein festes Datenschema, wie es beispielsweise bei SQL verwendet wird. Dabei wird für jede Datensatz-Klasse ein fester Satz von Attributen definiert. Eine Instanz einer solchen Klasse (ein Datensatz/Tupel) kann dann keine anderen Attribute, wie vorgegeben, besitzen. Für jedes Attribut ist zudem ein fest vorgegebene Datentypen festgelegt. Diese Organisation ist für die Zugriffszeiten vorteilhaft, verhindert jedoch heterogene Datenformate oder das vollständige Auslassen von Daten oder das Wiederholen von einem Attribut für ein Tupel (z.B. mehrere Bevölkerungszahlen für unterschiedliche Zeitpunkte für ein Population Attribut).

2.2.2. Semistrukturierte Daten

Unter semistrukturierten Daten versteht man eine Struktur, wie sie in XML oder RDF verwendet wird.

Man findet sie in vielen Bereichen, wie in der Bioinformatik, wo Daten in heterogenen Formaten zwischen Datenbanken oder sonstigen Anwendungen ausgetauscht werden und denen kein einheitliches Schema zugrunde liegt. Deswegen wurden solche Daten zunächst *unstrukturiert*, danach *semistrukturiert* genannt.

Die Daten sollen dabei eine Struktur besitzen, die mit den flachen Tupeln des relationalen Datenmodells nur unzureichend wiedergegeben werden kann. Auch das Objektmodell sei oft ungeeignet: Zwar könne man damit auch tiefe Strukturen repräsentieren, allerdings keine unregelmäßige Strukturen mit fehlenden oder wiederholten Komponenten. Fehle das Schema, so müsse zudem die Bedeutung der Struktur in den Datensätzen selbst wiedergegeben werden. (vgl. [3])

2.3. SQL

SQL[13] (kurz für: „Structured Query Language“) ist eine strukturierte Abfragesprache zur Erstellung und Benutzung von relationalen Datenbanken. SQL ist die momentane Standardanfragesprache in der Wirtschaft, da sie sich durch leichte Bedienbarkeit und hohe Geschwindigkeit auszeichnet. Es gibt zahlreiche Erweiterungen für diese Sprache wie z.B. PSQL, mit der sie auch in der Lage ist objektorientierte Datenbanken zu erstellen. Nachfolgend wird nur ein grober Überblick über die Anfragesprache SQL und das dazugehörige relationale Modell gegeben.

In der Anwendung werden relationale Datenbanken meist unter Zuhilfenahme von *Entity Relationship*-Diagrammen (kurz: ER - Diagramme) erstellt. Diese ER-Diagramme beschreiben die Struktur der Daten und ihre Zusammenhänge sowohl durch Angabe der Entitätstypen und der jeweils dazugehörigen Attribute als auch durch die Beziehungen zwischen den Entitätstypen und der jeweiligen Kardinalitäten. Aus diesem ER-Diagramm kann dann durch die entsprechende Umformung eine (SQL)-Datenbank erstellt werden, die die Datensätze in verschiedenen Tabellen (Relationen) speichert. Detaillierte Informationen über ER-Diagramme werden gezeigt in: [5]. In Abbildung 2.2 wird ein Ausschnitt des ER-Diagrams der Mondial-Datenbank gezeigt. Der Country-Entitätstyp besitzt viele

verschiedene Attribute, die später in der umgesetzten Country-Tabelle als Spalten dargestellt sind. Auch die Beziehung capital von Country mit dem Entitätstyp City wird als Spalte in dieser Tabelle abgelegt, in der als Referenz zu City der Primärkey, also der Name der Stadt, für jeden Datensatz von Country gespeichert wird. Beziehungen wie beispielsweise is_member oder borders, die keine 1:n Relationen sind und ein eigenes Attribut besitzen, müssen als eigene Tabelle mit n:m-Kardinalität gespeichert werden.

Die Datensätze, die in den Tabellen gespeichert sind, können dann mit der Anfragesprache SQL gefiltert und ausgegeben werden. Dabei benutzt SQL Schlüsselwörter, die jeweils einen anderen Bereich der SQL-Abfrage einleiten. Die wichtigsten Abschnitte sind hierbei der *SELECT-*, *FROM-* und *WHERE-*Abschnitt, man spricht auch vom SFW-Block. Jede SQL-Anfrage beginnt mit dem *Select-*Abschnitt, in dem festgelegt wird, welche Informationen (Spalten von Tabellen) nach Auswertung der restlichen Anfrage ausgegeben werden sollen. Danach kommt immer der *From-*Abschnitt, der angibt, welche Tabellen für die Anfrage benutzt werden sollen. Nun können mehrere optionale Abschnitte kommen, von denen der *Where-*Abschnitt der am häufigsten benutzte ist. Mit ihm können Bedingungen an die Anfrage gestellt werden, damit die Datensätze aus dem *From-*Abschnitt gefiltert werden. So kann zum Beispiel gefordert werden, dass die numerischen Werte einer Spalte größer oder kleiner einem bestimmten Wertes sein müssen oder die Werte, die in zwei Spalten von unterschiedlichen Tabellen stehen, gleich sein müssen. Auch Unterabfragen sind möglich, die es zum Beispiel mit dem Operator *Exists* ermöglichen zu überprüfen, ob der Wert einer Spalte in einer selbst erstellten Liste wiederzufinden ist oder nicht. Andere optionale Abschnitte sind *Order by*, der die ausgegebenen Datensätze nach den Werten in den angegeben Spalten sortiert, und *Group by*, der Datensätze zu Gruppen zusammenfasst, sodass Aggregatfunktionen auf eine Menge von Datensätzen angewandt werden können. In dem dazugehörigen *Having-*Abschnitt ist es mit diesen zudem möglich auch Bedingungen an die Gruppen zu stellen, ähnlich des *Where-*Abschnitts.

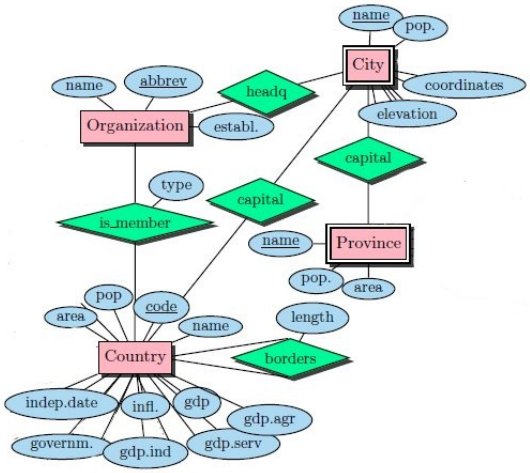

Abbildung 2.2.: Ausschnitt des ER-Diagrams der Mondial-Datenbank

Abbildung 2.3 zeigt eine SQL-Anfrage, in der die meisten der besprochenen Blöcke zum Einsatz kommen. Benutzt wurde nur die Tabelle City, wie sie oben aus dem ER-Diagram entsteht. Gesucht wurden alle Provinzen Deutschlands, deren Summe der Einwohner ihrer Städte 5 Millionen übersteigen. Das Ergebnis wird zudem nach Namen der Provinz sortiert.

Genauere Erläuterungen zu SQL-Anfragen in [4]

Wird eine solche Anfrage an eine Datenbank gestellt, so erstellt diese für die Auswertung den Syntaxbaum der Anfrage. Dieser wird so gut wie möglich durch verschiedene Techniken optimiert und danach ausgewertet. Die Ergebnisse werden dann an Variablen gebunden und ausgegeben.

.

```
SELECT Province
FROM City
WHERE country = 'D'
GROUP BY Province
HAVING Sum(Population > 5000000)
SORT BY Province
```

Abbildung 2.3.: SQL-Anfrage an die Mondial-Datenbank

Ein essenzielles Konzept bei der Auswertung von SQL-Anfragen ist die sogenannte *Closed-World-Annahme*. Die Auswertung negierter Bedingungen stellt normalerweise ein Problem für die Datenbank dar, da sie nur positive logische Schlussfolgerungen ziehen kann. Mit der *Closed-World-Annahme* geht die Datenbank davon aus, dass jegliche relevanten Informationen in der Datenbank gespeichert sind. Folglich kann für alle Datensätze, die eine positive Bedingung nach ihrer Auswertung nicht erfüllen, angenommen werden, dass sie also die Negation erfüllen. Diese Schlussfolgerung kann sich jedoch durch das Hinzufügen zusätzlichen Wissens hinterher als falsch herausstellen.

2.3.1. Joins

Ein wichtiger Aspekt bei Anfragen allgemein sind sogenannte *Joins*. Um den Unterschied zwischen den Joins bei SQL und SPARQL deutlich zu machen werden an dieser Stelle die Grundlagen der Anwendungen und Auswertung von Joins bei SQL erläutert und später im Bezug hierauf die essenziellen Unterschiede von SPARQL dargelegt. Für die Erklärung von Joins wird ein grundlegendes Verständnis von Mengenoperationen vorausgesetzt, die falls erforderlich in [6] nachgeschlagen werden können.

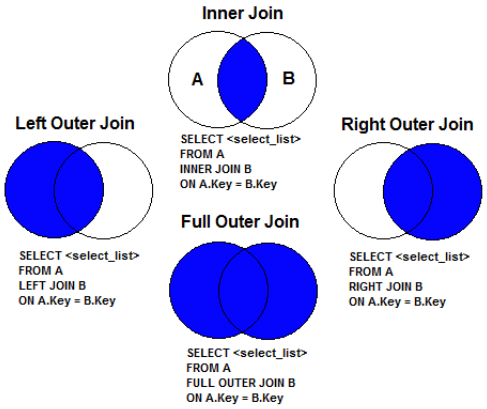

Abbildung 2.4.: Mengenselektion von verschiedenen Joins

Bei der Auswertung von SQL-Anfragen werden aus Tabellen oder Teiltabellen Datensätze selektiert. Für den Fall jedoch dass mehrere Tabellen verwendet werden, muss entschieden werden wie diese zusammengefügt werden. Hierbei kommen verschiedene Arten von Joins zum Einsatz, Abb.2.4 gibt einen Überblick über die geläufigsten Joins.

Der einfachste Join bei SQL ist der *„Cross Join“*, der implizit verwendet wird, wenn im *From*-Abschnitt zwei (oder mehrere) Tabellen mit einem Komma getrennt werden. Explizit kann auch statt dem Komma „cross join“ zwischen den Tabellen geschrieben werden. Der *cross join* stellt ein kartesisches Produkt der Tabellen dar, was bedeutet, dass jede Zeile (Datensatz) der eine Tabelle mit jedem Datensatz der anderen Tabelle kombiniert wird. Besitzt die Tabelle A 3 Spalten und 10 Einträge und Tabelle B 2 Spalten und 5 Einträge, so besteht die kombinierte Tabelle mit diesem Join aus allen 5 Spalten und 50 Einträgen. Dieser Join wird meistens verwendet, wenn es keine Verbindung über Fremdschlüssel zwi-

schen den Tabellen gibt. Wie man durch dieses kleine Beispiel sieht, wächst jedoch die Datenmenge bei diesem Join multiplikativ an und sollte aus Performance-Gründen bei sehr großen Tabellen nicht in Betracht gezogen werden.

Daher ist die bei SQL geläufigste Join-Art der *„inner-Join“*. Bei diesem Join wird angegeben, über welche Spalten die Tabellen verbunden werden, indem diese gleich gesetzt werden oder durch einen anderen Vergleichsoperator verbunden sind. Meistens wird dafür jeweils eine Spalte der Tabellen verwendet, es können aber auch beliebig weitere Verbindungen angegeben werden, um das Ergebnis weiter einzuschränken. Dies geschieht in einem speziellen *On*-Abschnitt oder simpler im *Where*-Abschnitt. Explizit wird dieser Join mit der Angabe von „inner join“ zwischen den Tabellen im *From*-Abschnitt eingeleitet, doch durch die Angabe der Bedingung kann implizit ebenfalls nur ein Komma verwendet werden. Rein praktisch betrachtet wird nun ein *cross Join* ausgeführt, bei dem durch die Bedingung(en) viele Datensätze ausgeschlossen werden können, sodass die Ergebnismenge deutlich kleiner ist. Häufig wird zum Beispiel gefordert, dass die ID in beiden Tabellen oder der Name einer Person identisch sein müssen, wodurch der Join nur Datensätze verbindet, die auch logisch zusammengehören. So kann eine Tabelle alle eindeutigen Informationen über gegebene Personen enthalten. Andere Tabellen speichern mit Referenz auf einen eindeutigen Datensatz in der ersten Tabelle Informationen, die häufiger in Bezug auf eine Person vorkommen können in einer 1:n Relation. Auf Bedarf können nun in einer Anfrage ein Teil dieser Informationen zusammengefügt werden und trotzdem bleibt die Anzahl der Datensätze in der Datenbank deutlich geringer, als alle Informationen gleichzeitig in einer Tabelle zu speichern.

Da am häufigsten für die Bedingungen bei einem *inner Join* Gleichheit verwendet wird, nennt man Joins mit nur solchen Verbindungen auch *equi-Joins*. Dieser kann zusätzlich zu den oben beschriebenen Möglichkeiten auch mit einem einfachen „join“ zwischen den Tabellen eingeleitet werden. Die Auswertung bleibt unverändert, doch sollte an dieser Stelle noch einmal darauf hingewiesen werden, dass SQL für Datensätze mit keinem Eintrag (auch NULL-Werte genannt) in einer Spalte *nicht* mit anderen Werten vergleichen kann. Sie fallen also automatisch bei der Filterung raus, da auch NULL = NULL nicht true ergibt.

Ein weiterer SQL Join ist der *natural Join*. Er wird mit dem Begriff „natural join“ zwischen den Tabellen eingeleitet und sucht in den Tabellen selbstständig nach Spalten mit gleichen Namen, um mit diesen einen *equi-Join* auszuführen.

Eine spezielle Art von Joins sind die *Outer Joins*. Diese unterteilen sich in *left-, right- und full outer Join*, von denen aber nur auf den *left outer Join* genauer eingegangen wird, da die anderen mit diesem leicht hergeleitet werden können.

Beim *left outer join* ist das Entscheidende, dass auf jeden Fall alle Datensätze der linken Tabelle in die Ergebnismenge übernommen werden. Prinzipiell wird ein *inner Join* durchgeführt, doch filtern die Bedingungen keine Datensätze der linken Tabelle. Findet sich für einen Datensatz der linken Tabelle kein entsprechender Datensatz in der rechten Tabelle, so werden in allen Spalten, die durch die rechte Tabelle dazukommen, keine Eintragungen vorgenommen. Sie enthalten also den NULL-Wert.

Der *right outer Join* ist äquivalent zu dem *left outer Join,* mit dem Unterschied dass alle Datensätze der rechten Tabelle auf jeden Fall ausgegeben werden. Der *full outer Join* ist eine Kombination aus *left outer join* und *right outer join*, da sie die Vereinigung der Ergebnismengen der einzelnen Joins darstellt.

2.4. RDF & SPARQL

Das Resource Description Framework [9] (RDF) wurde vom World Wide Web Consortium (kurz: W3C) als Standard zur Beschreibung von Metadaten entwickelt. RDF ist dabei ähnlich, wie das ER-Modell, nur ein Modell das konzeptuell festlegt, wie die Daten beschrieben werden können. Es ist dabei jedoch nicht strukturiert, sondern semistrukturiert. In der Anwendung werden meistens die Sprachen XML oder N3 verwendet, um das Modell textuell zu repräsentieren.

Abbildung 2.5.: RDF Icon (Quelle: http://www.w3.org/)

Heutzutage gilt RDF zudem als ein Grundbaustein des Semantic Webs, dessen Ziel es ist, den Informationen im Internet eine semantische Bedeutung zuzuordnen, damit weltweite Datenquellen besser miteinander kombiniert und ausgewertet werden können. Das RDF Modell stellt eine Art gerichteten Graphen dar, dessen Knoten und Kanten durch Tripel beschrieben werden. Damit bilden Tripel ein Grundelement eines RDF Modells und werden im dafür vorgesehenen Kapitel näher erläutert (s. auf der nächsten Seite).

Es ist möglich die Tripel eines RDF-Modells als solche in einer relationalen Datenbank ohne Verluste. mit verschiedenen Vorgehensweisen, zu speichern. Die Naivste wäre eine große Tabelle mit vier Spalten anzulegen, wobei zwei Spalten mit Datentyp VarChar das Subjekt und die Property speichern, während das Objekt mit Typ binary large object (blob) abgelegt wird. Die vierte Spalte speichert dann von welchem Datentyp das Objekt ist. Diese Lösung bringt aber einige Probleme mit sich, so können zum Beispiel Einträge nicht in Spalten vom Typ blob gesucht werden. Alle Vorgehensweisen haben zudem das Problem gemeinsam, dass die Suche nach spezifischen Attributen sehr ineffizient ist (s. [11]). Die Möglichkeiten der relationalen Speicherung werden in einer solchen Struktur nicht ausgenutzt und SQL Datenbanken sind nicht dafür ausgelegt, riesige Datenmengen in einer einzelnen Tabelle zu speichern.

Daher wurden für die Speicherung der Triples spezielle Triplestores entwickelt. Alternativ wäre es auch möglich die Datensätze im XML-Format in einer (oder mehreren) Dateien abspeichern, was allerdings für größere Datensätze problematisch wird.

Analog zu SQL gibt es auch für Datenbanken, die Daten nach dem RDF-Modell abspeichern, eine Anfragesprache um gewünschte Datensätze nach der Speicherung wiederzufinden. Die W3C empfiehlt hierfür seit Januar 2008 die Sprache SPARQL (**S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage).[15]

2.4.1. Jena & ARQ/SPARQL

Die Anfragesprache SPARQL kann über das Anfragetool Apache Jena[7] mit der ARQ Engine verwendet werden. Die Verwendung des Programms wird, wie in [8] näher beschrieben, durchgeführt.

Die grundsätzliche Struktur von SPARQL-Anfragen ähnelt derer von SQL, denn auch hier wird der SFW-Block verwendet. Da die Anfrage aber nicht nach Einträgen in Tabellenspalten sucht, sondern nach Tripeln, oder besser gesagt nach Kanten in einem Graphen, baut SPARQL auf Variablen auf, die die gewünschten Ergebnisse an sich binden. Diese

Variablen sind entscheidend für die Konvertierung, daher wird in dem Kapitel Variablen noch einmal näher auf sie eingegangen.

2.4.2. Tripel

Das *Tripel* bildet die Grundlage der Datenspeicherung in RDF, denn alle Daten einer RDF-Datenbank sind in diesem Format abgelegt. Die Struktur des *Tripels* ist dabei immer „Subjekt - Prädikat - Objekt“, die im Folgenden weiter erläutert werden. Bildlich gesprochen baut das *Tripel* eine Beziehung zwischen Subjekt und Objekt auf, also wie eine Kante in einem Graphen. Diese Beziehung ist vom Subjekt auf das Objekt gerichtet und durch das Prädikat benannt.

Das Subjekt ist normalerweise immer ein Objekt der Welt, über das Informationen gespeichert werden sollen. Diesen Objekten werden durch *Unified Resource Identifier* (im weiteren URI) eindeutige Namen gegeben und sie damit angesprochen, sodass zum Beispiel zwei Personen mit gleichen Namen auch unterschieden werden können. Auf diese Art und Weise lassen sich verschiedene Datenquellen einfach miteinander verbinden, sofern für die selben Objekte der Welt die gleichen URIs benutzt werden. Eine URI ähnelt zudem meist der Form einer URL, muss dies aber nicht. Betrachten wir zum Beispiel das Land Deutschland aus der Mondial-Datenbank, so wird es durch die URI:
<http://www.semwebtech.org/mondial/10/countries/D/> beschrieben. Die URI des Bodensees ist entsprechend: <http://www.semwebtech.org/mondial/10/lakes/Bodensee/>. Solche URIs in Form einer URL müssen nicht unbedingt auch über das Internet erreichbar sein.

Prädikate werden ebenfalls durch eine URI beschrieben und verbinden das Subjekt mit dem Objekt durch eine Eigenschaft (oder Property). Mögliche Properties wären zum Beispiel
<http://www.semwebtech.org/mondial/10/meta#name> oder
<http://www.semwebtech.org/mondial/10/meta#population>. Damit immer gleich bleibende Präfixe wie <http://www.semwebtech.org/mondial/10/meta#> nicht immer ausgeschrieben werden müssen, können sie am Anfang durch einfache Zeichenfolgen ersetzt werden. Eine solche Definition wird mit „PREFIX *Ersetzung Präfix*“ durchgeführt.

Das Objekt des Tripels ist allerdings nicht unbedingt ebenfalls eine reale Sache, die durch eine URI beschrieben werden kann, sondern kann auch ein Literal darstellen. So sind also auch Zeichenketten, Zahlen, Wahrheitswerte etc. möglich, die natürlich durch eine Datentypangabe interpretiert werden müssen.

Zusammengenommen könnten ein paar *Tripel* nun wie folgend aussehen:

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
<http://www.semwebtech.org/mondial/10/countries/GR/> rdf:type :Country .
<http://www.semwebtech.org/mondial/10/countries/GR/> :name 'Greece'.
<http://www.semwebtech.org/mondial/10/countries/GR/> :carCode 'GR'.
```

Abbildung 2.6.: Tripel Beispiele

Hier wird dem Subjekt Griechenland, der Name „Greece“ und das Länderkürzel „GR“

zugeordnet. Zudem wird es über die Property rdf:type (auch kurz geschrieben „a“) als Objekt der Klasse Country definiert.

Auch bei *SPARQL*-Anfragen spielen *Tripel* eine wichtige Rolle, da durch ihre Angabe bestimmt wird, welche *Tripel* der Datenbank gefunden werden sollen. Dafür werden Variablen eingesetzt.

2.4.3. Variablen

Variablen ersetzen in SPARQL-Anfragen Subjekte, Prädikat und/oder Objekte der gesuchten Tripel. Die Variablenbindungen sind bei SPARQL nicht wie bei SQL-tabellenorientiert, sondern stehen für eine Menge von Objekt mit gemeinsamen Eigenschaften. Deshalb werden sie im Weiteren als SPARQL Variable (kurz SPARQL-Var) bezeichnet. Diese Variablen definieren sich aus der den Informationen der Tripel und werden in den BGPs zusammengeführt. Je mehr Informationen also zu einer SPARQL-Var bekannt sind, desto stärker grenzt sich ihre Menge ab und desto präziser ist sie definiert. Eine SPARQL-Var kann ein Attribut, eine einzelne Klasse oder auch mehrere Klassen beschreiben. SPARQL-Var sind für die Anfrage global gebunden und sollten sie mehrfach zugewiesen werden, so bedeute dies einen Join der beiden Mengen.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?n
WHERE { ?x a :City.
        ?x :name ?n. }
```

Abbildung 2.7.: Simple SPARQL-Anfrage

Abb.2.7 ein simples Beispiel einer SPARQL-Anfrage. Gesucht sind alle ?x und ?n, die die Bedingungen der Where-Klausel erfüllen. Durch die erste Bedingung wird bestimmt, dass ?x nur Ergebnisse der Klasse „City“ binden kann. Die zweite Bedingung gibt an, dass es für alle ?x ein Tripel mit dem Prädikat : *name* dessen Wert an ?n gebunden wird in der Datenbank geben muss. Es werden also für alle Städte in der Datenbank ihre URI und ihre Namen ausgegeben. Da ?n nur als Objekt in allen Bedingungen vorkommt, bezeichnen man es als ein Literal. Ohne die explizite Angabe, dass ?x eine City ist, müsste man davon ausgehen, dass alle Klassen mit einem : *name* Attribut gesucht werden. Folglich würden z.B. auch alle Seen, Organisationen und Länder ausgegeben werden.

2.4.4. Filter

Die Filterung ist einer der wichtigsten Abschnitte. Er ermöglicht das Stellen von Filterbedingungen und ist die SPARQL-Repräsentation des Algebraausdruckes σ. In Abb. 2.8 wird eine erweiterte Anfrage von Abb. 2.7 gezeigt, mit der nur noch alle Städte außer Berlin gesucht werden. Die Angabe von gesuchten Tripeln in der Anfrage, wie oben beschrieben, entspricht natürlich schon einer Art Filterung der Datensätze, daher sind diese auch im *Where*-Block gelistet. Der Filter-Block wird daher eher verwendet um Variablenbindungen mit Konstanten zu vergleichen oder Ungleichheit zwischen zwei SPARQL-Vars zu fordern.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?n ?t
WHERE { ?x a :City.
        ?x :name ?n.
        FILTER ( ?n != 'Berlin') . }
```

Abbildung 2.8.: SPARQL-Anfrage mit Filter

Der Inhalt der *Filter*-Bedingung in einer SPARQL-Anfrage entspricht dann eher dem *Where*-Block einer SQL-Anfrage. Alle gängigen Operatoren stehen für die Einschränkung der benutzten SPARQL-Vars zur Verfügung und sie können „beliebig“ komplex werden. Anders als die in SQL bekannten Unterabfragen sind diese jedoch im *Filter*-Abschnitt nicht möglich und auch nicht nötig, da Unterabfragen in SPARQL einfach durch die Verwendung neuer SPARQL-Vars ausgeführt werden können.

2.4.5. Optional

Der Optional-Abschnitt ist für Variablenbindungen, die nicht unbedingt erfüllt werden müssen, zuständig. Ein Tupel ist auch dann noch Teil der Ergebnismenge, wenn für die SPARQL-Vars im Optional keine entsprechenden Werte gefunden werden. Sie binden Werte für Tupel, bei denen es einen entsprechenden Wert gibt, ansonsten wird der Null-Wert gejoint. Optional-Abschnitte sind daher in SPARQL die einzige Möglichkeit Null-Werte zu erzeugen.

Da die Speicherung in RDF semistrukturiert ist, können zum Beispiel einzelne Objekte (Datensätze) einer Klasse, die durch eine SPARQL-Var selektiert wird, zusätzliche Attribute im Vergleich zu den restlichen Objekten dieser Klasse besitzen. Für diese kann dann eine zusätzliche Spalte gejoint werden, während für die restlichen Objekte keine Wertebindungen durchgeführt werden kann. In Abb. 2.9 wird ein solcher Fall dargestellt. Die SPARQL-Var $?x$ bindet nur Werte der Klasse City. Diese besitzen alle ein Name-Attribut, doch muss nicht jede Stadt an einem Gewässer liegen und besitzt dadurch das locatedAt-Attribut nicht. Durch die dargestellte Anfrage werden die Namen von allen Städten ausgegeben und zusätzlich, sofern vorhanden, das Gewässer an denen sie liegt.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?n ?w
WHERE { ?x a :City.
        ?x :name ?n.
        Optional{ ?x :locatedAt ?w }
      }
```

Abbildung 2.9.: SPARQL Anfrage mit Optional

In Abb.2.10 wird gezeigt, wie *Optionals* zusammen mit Filtern auch als negative Existenzquantoren[1] verwendet werden können. Dabei wird im *Optional* versucht die SPARQL-

[1] In der aktuellen Version von SPARQL ist auch ein (not) Exists möglich, allerdings noch nicht in der für diese

Var zu binden [2], aber dann im Filter die Bedingung gestellt, dass kein Ergebnis gebunden wurde. In diesem Fall würde es bedeuten, dass alle Städte, die an einem Gewässer liegen, aus der Ergebnismenge entfernt werden.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?n
WHERE  { ?x a :City.
         ?x :name  ?n.
         Optional{ ?x :locatedAt ?w } .
         FILTER ( !BOUND(?w) ) . }
```

Abbildung 2.10.: SPARQL-Anfrage mit Not Exists

2.4.6. Joins

In SPARQL werden Joins durch das Binden von der selben SPARQL-Var in verschieden Tripeln erzeugt. Wird eine Variable in einem Tripel (s. auf Seite 14) verwendet und im selben BGP auch in einem anderen Tripel, so impliziert dies einen *equi-Join*. In Abb.2.11 ist ein Beispiel gezeigt bei dem einen Join über die SPARQL-Var ?*c* erfolgt

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?cn ?xn
WHERE {

    ?x a :Country.
    ?x :name ?xn.
    ?x :capital ?c.
    ?c :name ?cn. }
```

Abbildung 2.11.: Einfacher Join

Ist durch den Operator „Optional“ eine solche Verbindung getrennt, entsteht ein Left Outer Join. Wie z.B. in Abb.2.9 wo über die SPARQL-Var ?*x* gejoint wird. Der Right - oder Full Outer Join sind in SPARQL nicht möglich.

Wie im SQL-Kapitel schon erklärt, sind in SQL keine Joins mit NULL-Werten möglich. Dies ist in SPARQL aber möglich, wenn eine SPARQL-Var an unterschiedlichen Stellen gebunden werden kann. Da nur durch die Verwendung von „Optional“ NULL-Werte in SPARQL entstehen können, kommt diese Besonderheit auch nur vor, wenn die Tripel, in denen die SPARQL-Var gebunden wird, in unterschiedlichen Optional-Blöcken sind. Bei der Konvertierung zu SQL wird für solche Fälle eine Lösung vorgestellt, die eventuelle NULL-Werte beim Join berücksichtigt (s. auf Seite 11). In Abb.2.12 ist ein Beispiel für diesen Fall. Es werden alle deutschen Städte aufgelistet, die entweder am selben Fluss, keinem

Arbeit verwendete Jena Version

[2]Der BOUND Operator ist im Query Converter nicht behandelt und kann deshalb nicht in Anfragen übersetzt werden, auch wenn dies technisch machbar gewesen wäre, ist es aus Gründen des Umfangs nicht gemacht worden.

Fluss oder nur eine an einem Fluss, folglich alle Städtekombinationen, die nicht an verschieden Flüssen liegen. Es werden jeweils normale Joins über ?$C1$ und ?$C2$ ausgeführt, da sie außerhalb eines Optionals in einem Triple vorkommen. Für die Variable ?W ist dies aber nicht der Fall, da es nur in zwei verschiedenen Optionals vorkommt. Deshalb kann der Fall vorkommen, dass es zweimal an unterschiedliche Werte gebunden wird. Diese sind dann nicht Teil der Ergebnismenge, da sie durch die Verwendung der selben SPARQL-Var gleich sein müssen. Bindet nur einer der beiden Optional-Blöcke einen Wert, so wird dieser Wert übernommen. Wird es weder in dem einem Optional, noch in dem Anderen gebunden, so ist dieser Datensatz immer noch Teil der Ergebnismenge und wir erhalten in diesem Fall den Join über NULL-Werte.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?C1 ?W ?C2
WHERE{

    ?C a :Country.
    ?C :hasCity ?C1 .
    ?C :hasCity ?C2 .
    ?C :carCode ?CC .
    OPTIONAL { ?C1 :locatedAt ?W } .
    OPTIONAL { ?C2 :locatedAt ?W } .
    FILTER (?CC = 'D')

}
```

Abbildung 2.12.: test-join-mit-nulls.sparql

2.4.7. Group by

Mit dem Group by-Schlüsselwort können, wie in SQL, Gruppen von Datensätzen zusammengefasst werden. Dies wird benötigt, um Aggregationsfunktionen, wie *min*, *max* oder *count*, anwenden zu können. Nach dem Group by können nur noch die SPARQL-Vars über die gruppiert wird und Aggregationsfunktionen auf die restlichen SPARQL-Vars zur Ausgabe verwendet werden. Die Abb.2.13 zeigt eine Anfrage, mit der alle Länder, die mehr als 100 Städte in der Datenbanken haben, gesucht werden.

Das Group by in der verwendeten Jena Version ist leider noch nicht vollständig umgesetzt. Prinzipiell ist es mit SPARQL zwar möglich sowohl über eine HAVING Klausel mit Aggregationsfunktionen Bedingungen an die Gruppierung zu stellen, als auch Aggregationsfunktionen im *Select*-Abschnitt auf SPARQL-Vars anzuwenden. Jedoch bietet die verwendete Version von Jena nur die Möglichkeit der HAVING Klausel. Aus diesem Grund wird auch nicht mehr weiter auf die Selektierung der Aggregationsfunktionen eingegangen.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?c
WHERE {

    ?c a :Country .
    ?c :hasCity ?x

}
Group by(?c)
Having(count(?x) > 100 )
```

Abbildung 2.13.: GroupBy-simple.sparql

2.4.8. Reified und symmetrische Reified-Beziehungen

Bei Reified- und SymmReifiedbeziehungen handelt es sich um n:m Beziehungen, die über ein Attribut oder mehrere Attribute, die für diese Beziehung typisch sind, verfügen. In SQL kann hierfür eine gewöhnliche n:m Tabelle einfach um entsprechenden Spalten erweitert werden. Im RDF Model gibt es diese n:m Tabellen nicht, sonder die Entität hat einen Verweis auf eine Andere in Form der URI. Diesem Verweis können keine zusätzlichen Informationen gegeben werden. Deshalb erstellt man eine neue Entität, die neben den URIs der beiden in Beziehung zusetzenden Entitäten, noch die weiteren Attribute der Beziehung enthält.
Der Unterschied zwischen einer gewöhnlichen und einer symmetrischen Reified-Beziehung besteht in der Art der Entitäten. Die gewöhnlich verbindet zwei verschiedene Klassen miteinander, in Mondial ist die Is_Member-Beziehung von Country und Organization mit dem Attribut Typ, eine refizierte Beziehung. Die Beziehung von Border in Mondial, fällt allerdings nicht in diese Klasse, obwohl diese Tabelle eine n:m Beziehung mit der Länge als zusätzliches Attribut hat, verweist sie nicht auf zwei Unterschiedliche Klassen, denn Beide sind vom Typ Country. Dies ist eine sogenannte SymmReified oder symmetrisch refizierte Beziehung, sie stellt also eine Beziehung zwischen zwei Entitäten her, die die selbe Klasse haben, her. Die Positionen der Einträge sind somit also vertauschbar, deshalb müssen entweder beide Varianten gespeichert werden, oder die Anfragen immer gegen beide Möglichkeiten.

3. RDF2SQL

Dieses Programm konvertiert in mehreren Schritten, basierend auf die in [10]vorgestellte Methode, eine RDF-Datenbank in eine SQL Datenbank. Zur Speicherung der Datenbank wird in dieser Arbeit Oracle SQL[13] verwendet. Dafür wird eine Kette von SPARQL-Anfragen verwendet, die unter Zuhilfenahme des RDF-Schemas essenzielle Informationen für die Umwandlung bereit stellt. Zu diesen Informationen zählen zum Beispiel die Identifikation von funktionalen Properties und das korrespondierende Relationale Schema. Dieser Schritt wurde exakt von [10] übernommen und kann dort genauer nachvollzogen werden (s. [10] Seite 52).

Aus diesen Informationen werden die Tabellen der SQL-Datenbank erstellt. Danach werden die Datensätze aus den Tripeln zusammengestellt und eingefügt. Anstatt eine komplette Datenbank umzuwandeln, ist es ebenfalls möglich einzelne Tripel einer bestehenden Datenbank hinzuzufügen.

Genauere Erläuterungen zur Implementierung sind in [14] und Details zu der Gewinnung der Metadaten in [10] zu finden.

3.1. Metainformationen

Die vorgestellten Tabellen dienen hauptsächlich zur Vorbereitung der Konvertierung von SPARQL- zu SQL-Anfragen, die ohne diese überhaupt nicht möglich wäre. Diese Abhängigkeit impliziert, dass RDF2SQL unbedingt vor dem Gebrauch des Query Converters ausgeführt werden muss.

3.1.1. Mapping Dictionary

Das Mapping Dictionary ist der wesentliche Unterschied zu einer normalen SQL Datenbank, da es ermöglicht auf die Knowledgebase des zugrundeliegenden RDF Modells zuzugreifen.

CLASS	PROPERTY	RANGE	TABLENAME	LOOKUPATTR	INV
Country	name	VARCHAR2(200)	Country	name	false
Country	capital	VARCHAR2(200)	Country	capital	false
City	isCapitalOf	VARCHAR2(200)	Country	capital	true
City	population	NUMBER	City	population	false
Border	bordering	VARCHAR2(200)	Border	Country1	false
Border	bordering	VARCHAR2(200)	Border	Country2	false

Tabelle 3.1.: Auszug aus dem Mapping Dictionary

"Im *Mapping Dictionary* (s.[10] Seite 17) kann nachgeschlagen werden, in welcher Tabelle und Spalte eine Eigenschaft *p* eines *RDF Objekts o* gespeichert ist."[14] Im Grunde entspricht das Mapping Dictionary einer HashMap, die für eine Kombination aus Klasse und geforderte Property des Tripels auf die entsprechende SQL-Tabelle und Spalte verweist, in der die Informationen der Property abgelegt sind. Anfangs war eine solche eindeutige Zuweisung vorgesehen, doch es stellte sich heraus, dass für das Mapping von SymmReified-Properties (s. auf Seite 19) schließlich zwei Spalten einer Tabelle infrage kommen. In der Mondial-Datenbank gilt dies nur für die Klasse Border und ihre bordering-Property. Sie verweist gleich zweimal auf die Border-Tabelle, die sowohl mit der Spalte Country1 als auch mit Country2 auf die Tabelle Country referenziert. Diese einzigartige Eigenschaft kann also dazu genutzt werden um entsprechende SymmReified-Properties zu erkennen und richtig umzuwandeln. Ebenfalls wird über die Spalte INV angegeben, ob die Property eine Inverse einer anderen Property ist. In diesem Beispiel sieht man, dass die Property isCapitalOf der Klasse City das Inverse der Property capital der Country-Klasse sein muss. Eine solche Verbindung kann in der Inverses-Tabelle nachgeschlagen werden. Solche Informationen werden benötigt, damit die Tabellen bei einem Join richtig verbunden werden können.

Eine weitere Aufgabe des MD ist es mit inversen Properties um zu gehen. Dies passiert wenn die gesuchte Property als Inverses einer anderen Property gespeichert ist. Denn zumindest theoretisch kann zu jeder Property, die zwei URIs miteinander verbindet (man bedenke, dass das Subjekt immer eine URI sein muss), eine inverse Property definiert werden. Anbieten tut sich dies aber nur, wenn es sinnvoller oder angenehmer ist, die andere Seite der Kardinalität der Beziehung zu benutzen, wie z.B. bei 1:n-Beziehungen. Solange die inverse Property im RDF-Schema definiert ist, kann diese auch in Anfragen verwenden werden, selbst wenn diese Richtung nicht explizit in der ursprünglichen RDF-Datenbank verwendet wurde. Solche Einträge werden dann in der Inv-Spalte des MD gekennzeichnet (in Abb.3.1 ist der dritte Eintrag ein Inverses) und es kann die nicht inverse Property mit der Inverse-Tabelle gefunden werden, die alle Paarungen von inversen und den zugehörigen nicht inversen Properties abspeichert.

3.1.2. NMJoinAttrs Table

CLASS	TABLENAME	LOOKUPATTR	FKJOINATTR
Island	locatedIn	Area	Entity
Sea	locatedIn	Area	Entity
City	locatedAt	Water	City
Lake	locatedAt	City	Water
Sea	locatedAt	City	Water
River	locatedAt	City	Water
Country	is_Member	Country	isMemberof

Tabelle 3.2.: Auszug aus der NMJoinAttr-Tabelle

Für Tabellen, die eine n:m-Beziehung zwischen zwei Entitätstypen speichern, reicht das Mapping Dictionary alleine nicht aus. Eine solche Tabelle besteht meistens nur aus zwei

Spalten, die die Primärschlüssel, der beiden Tabellen in Beziehung setzt. Manchmal wird noch eine zusätzliche Information abgespeichert, wie z.B. in der Tabelle is_Member, die nicht nur speichert, welche Länder Mitglieder in welcher Organisation sind, sondern auch welchen Typ diese Mitgliedschaft besitzt.

Soll nun in einer Anfrage eine Verknüpfung der beiden Tabellen über eine solche Hilfstabelle entstehen, so muss von einer Richtung aus festgestellt werden, welche Spalte für die andere Richtung zuständig ist. Beispielsweise wenn man für eine *Country* die *Organizations* wissen möchte, so muss ein Join über die n:m Tabelle *Membership* gemacht werden. Dazu wird nach dem LookUpAttr des MD und der Klassen der CV gesucht in der NM Tabelle suchen. Für das Beispiel sucht man also mit der Klasse *'Country'* und aus dem MD mit dem Tablename *'Membership'*. Damit erhält man sowohl den Namen der Spalte des LookUpAttr in der NM Table, als auch den Namen des anderen Attributes der n:m-Beziehungn, das FKJoinAttr. Damit kann nun die referenzierte Tabelle dieser Relation gefunden werden und alle Informationen für den n:m Join sind verfügbar.

Für das Beispiel bedeutet das nun, dass das LookUpAttr *ofMember* haben und das FKJoinAttr mit dem nun wiederum im MD nachgeschlagen werden kann welche zweite Tabelle nötig ist.

3.1.3. RangeClassJoin Table

Gleicherweise ist es für Tabellen, die weitere URI-Spalten speichern und damit auf eine andere Tabelle referenzieren, notwendig zu wissen, was das Ziel dieser Referenzierung ist. Hierfür wird die RangeClassJoin Tabelle verwendet, in der alle Verknüpfungen über Fremdschlüssel hinterlegt sind. Für die Anfragenkonvertierung wird dies genutzt, wenn ein Join über eine solche Spalte benötigt wird. Eine einfache Kombination aus Tabellen- und Spaltenname liefert den Namen der Zieltabelle auf deren URI-Spalte verwiesen wird.

TABLENAME	COLUMNNAME	RANGETABLE
Border	Country1	Country
Border	Country2	Country
locatedAt	Water	Sea
locatedAt	Water	Lake
locatedAt	Water	River
Country	Capital	City

Tabelle 3.3.: Auszug aus der RangeClassJoin-Tabelle

Wie noch einmal in der obigen Tabelle zu sehen ist verweisen bei SymmReified Tabellen beide reifizierten Spalten auf die selbe Hometabelle. Durch Anfrage an die RangeClassJoin-Tabelle wird so beispielsweise auch erkannt, dass die Spalte Water von locatedAt auf alle Gewässer verweist. Dies bedeutet, dass bei einem Join nicht nur die Tabelle Sea betrachtet werden muss, sondern auch River und Lake.

3.2. Relationales Modell erstellen

3.2.1. Erstellung der Hometables

Die Hometables sind alle Tabellen, die eine entsprechende Klasse des RDF-Schemas widerspiegelt. So gibt es für jede Klasse, wie z.B. Country oder Sea, eine gleichnamige Tabelle, die die funktionalen Properties für diese Klasse speichern.

Um diese Tabellen zu erstellen, wird die Anfrage aus Abb. 3.1 verwendet. Mit dem Ergebnis kann das Mapping Dictionary (s. auf Seite 20) gefüllt werden. ?*C* ist die Klasse, ?*P* die Property und ?*R* die Range des Eintrages.

Falls ?*R* eine *RDF-Klasse* ist so muss ebenfalls ein Eintrag in der *RangeClassJoin-Tabelle* hinterlegt werden.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX aux: TAG_aux
SELECT DISTINCT ?C ?P ?R
FROM <TAG_Source-rel-schema.n3>
FROM <TAG_Source-class-props.n3>
WHERE {

    ?C aux:hasFunctProp ?P .
    ?A aux:onDomain ?C ;
       aux:onProperty ?P ;
       aux:onRange ?R

}
```

Abbildung 3.1.: schema-funct.sparql

3.2.2. Erstellung der NM Tabellen

Für die NMTabellen gelten andere Bedingungen als für die *Hometables*, deshalb müssen sie mit einer eigenen Abfrage gefunden werden. In den Metainformationen sind bereits alle Tabellen gespeichert die NM Relationen gespeichert und müssen nur noch mit der Anfrage aus Abb.3.2 erfragt werden und entsprechende Tabellen mit den richtigen Spalten erstellt werden (s.[10] Seite 68).

Für die entsprechenden Mapping Dictionary Einträge muss eine weitere SPARQL Anfrage (Abb. 3.3), in jeweils beide Richtungen, gestellt werden. Somit wird für jede mögliche *RDF-Klasse* der entsprechende Eintrag im *Mapping Dictionary* hinterlegt. Zu beachten ist, dass für eine NM Tabelle durchaus eine Spalte mehrere Klassen beinhalten kann oder mit Oberklassen, wie z.B. Water, belegt sein kann und in der SQL-Anfrageumsetzung letztendlich mehrere Tabellen gejoint werden müssen, um diese Spalte zu erhalten.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : TAG_Colon PREFIX aux: TAG_aux
SELECT DISTINCT ?P ?D1 ?D2 ?P2
FROM <TAG_Source-rel-schema.n3>
FROM <TAG_Source-class-props.n3>
FROM <TAG_Source-er.n3>
FROM <TAG_Source-meta.n3>
WHERE {
    ?A :nmName ?P ;
       :nmOnDomain1 ?D1 ;
       :nmOnDomain2 ?D2 .
    OPTIONAL { ?P owl:inverseOf ?PINV } .
    OPTIONAL { ?P a owl:ObjectProperty .
               ?A :nmName ?P2 }
}
```

Abbildung 3.2.: schema-nm-object.spraql

```
prefix :        TAG_Colon
prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
prefix xsd:     <http://www.w3.org/2001/XMLSchema#>
prefix owl:     <http://www.w3.org/2002/07/owl#>
prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix aux:     TAG_aux
prefix er:      TAG_er
SELECT ?X ?Y ?O
FROM <mondial-rel-schema.n3>
FROM <mondial-er.n3>
FROM <mondial-meta.n3>
WHERE {
    ?A :nmName ?X .
    ?A :nmOnDomain1 ?Y .
    ?A :nmOnDomain2 ?Z .
    ?O rdfs:subClassOf ?Y ;
       er:isa er:Concrete
}
```

Abbildung 3.3.: nm-Dic.sparql

3.2.3. Entfernen der inversen Beziehungen

In einem RDF Model sind die Kanten der Beziehungen der Entitäten untereinander gerichtet, werden aber häufig für beide Seiten definiert. So hat beispielsweise eine Stadt ihre Flüsse als Kante in einer *„locatedAt“*-Beziehung und ein Fluss die Städte in einer *„flows“*-Beziehung. Durch die Semistrukturiertheit können solche Verbindungen auch für NM Beziehungen gemacht werden. In SQL muss eine NM Beziehung durch eine extra Tabelle konstruiert werden, dadurch spielt dann die Betrachtungsrichtung keine Rolle mehr und bei einer 1:1 Beziehung wird das Attribut in der selben Tabelle gespeichert. Aus diesem Grund sollte man die Inversen löschen, schon alleine um die Datenbank nicht unnötig aufzublähen und Redundanzen zu vermeiden. Dazu wird die SPARQL Abfrage in Abb.3.4 verwendet.

```
PREFIX : TAG_Colon
PREFIX er: TAG_er
PREFIX aux: TAG_aux
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT DISTINCT ?C ?P ?D ?PINV
from <file:TAG_Source-meta.n3>
from <file:TAG_Source-invdefs.n3>
from <file:TAG_Source-functional-props.n3>
from <file:TAG_Source-11props.n3>
WHERE {
    { { ?C aux:has11PropTotalInvTotal ?P }
    UNION
    { ?C aux:has11PropTotalInvTotal ?P } } .
    ?P owl:inverseOf ?PINV .
    ?D aux:hasFunctProp ?PINV .
    FILTER (str(?P) < str(?PINV))
}
```

Abbildung 3.4.: remove.sparql

3.2.4. SymmReified - Tables

Unter Reified Tables versteht man Tabellen mit Beziehungen, die neben den Fremdschlüsseln noch für die Beziehung spezifische Attribute haben, wie z.B. *Borders* neben den jeweiligen angrenzenden Ländern noch die Länge der Grenze speichert. Symmetrisch ist eine Tabelle, wenn Wertepaare in beiden Varianten vorhanden sind, wie beispielsweise sowohl Country1 = „Germany“, Country2 = „Austria“, als auch Country1 = „Austria“ und Country2 = „Germany“ in der Tabelle vorhanden ist. Daraus folgt dann auch, dass bei dieser Beziehung, beide Fremdschlüssel von der selben Klasse sein müssen. Es handelt sich dabei also um eine besondere Form der NM Tables.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : TAG_Colon PREFIX aux: TAG_aux
prefix er: TAG_er
SELECT ?R ?P ?C
FROM <TAG_Source-rel-schema.n3>
FROM <TAG_Source-class-props.n3>
FROM <TAG_Source-er.n3>
FROM <TAG_Source-meta.n3>
where {

    ?R er:isa er:SymmetricReifiedRelationship .
    ?R er:reifies
    ?Q. {
        { ?R rdfs:subClassOf ?X }
        union
        { ?R owl:equivalentClass ?X }
    } .
    ?P rdfs:domain ?X .
    ?P rdfs:range ?C .
    ?Q rdfs:domain ?C .
    ?Q rdfs:range .
    ?C FILTER (
        ?P != owl:bottomObjectProperty &&
        ?Q != owl:bottomObjectProperty
    )

}
```

Abbildung 3.5.: schema-sym-reified.sparql

Mit der Anfrage in Abb.3.5 wird festgestellt, für welche Tabellen diese Eigenschaft zutrifft und diese werden dann durch SymmReifeid Tabellen ersetzt. Trotzdem ist es notwendig, dass die Information welches ursprüngliche Attribut sie ersetzt verfügbar ist, damit

die Anfragen noch adäquat umgesetzt werden.

Damit die Anfragen auf diese Tabellen korrekt funktionieren, müssen die Einträge ins Mapping Dictionary besonders gestaltet werden. Normalerweise würde der Tabellenname auf die NM Tabelle des Attributes verweisen, welches im RDF Model die NM-Verbindung ist. Bordering allerdings enthält nur eines der NM Objekte und es wäre eine Kette von Country zu Bordering zu Border zum zweiten Bordering und zur zweiten Country notwendig. Da jedoch der Reifiedeintrag ebenfalls die beiden Referenzen enthält, können stattdessen auch in ihm beide NM-Einträge gefunden werden.

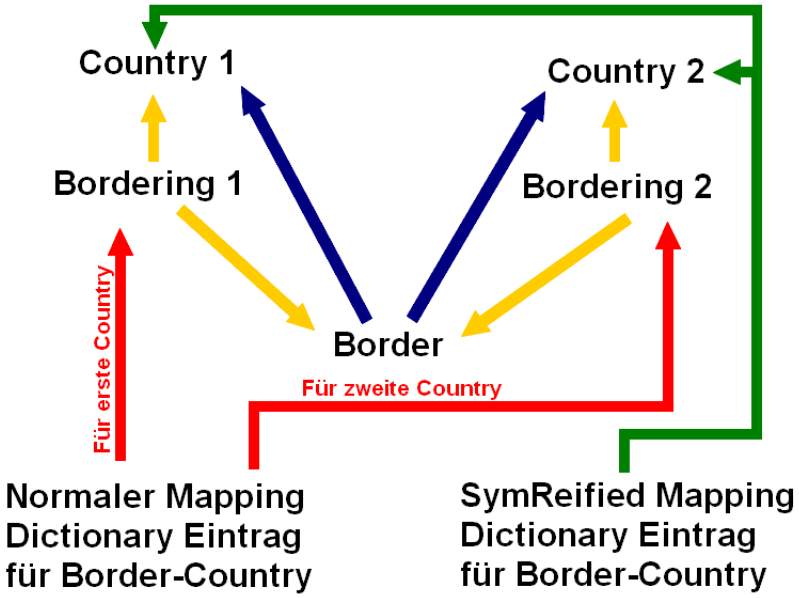

Abbildung 3.6.: Beziehungen im Mapping Dictionary bei SymmReified Tables. Die Pfeile stellen die jeweils gerichtete Beziehungen dar. Ziel ist es über den Mapping Dictionary-Eintrag mit Klasse Border und der Property bordering eine Verbindung zu Country zu bekommen. Mit den konventionellen Mapping Dictionary-Einträgen würde man für jede einzelne Country jeweils auf eine Bordering-Tabelle verwiesen werden (rote Linie). Es wäre möglich, dann über Bordering jeweils Country und Border als NM - Fremdschlüssel zu erhalten, allerdings ist dann immer noch nicht klar welche SPARQL-Var welcher Tabelle zugehörig ist (gelbe Linie). Aus diesem Grund werden spezielle Einträge für SymmReified Tables, im Mapping Dictionary gemacht (grüne Linie). Da Border auch eine Referenz auf der entsprechende Country hat, braucht es keine NM Tabelle dazwischen (blaue Linie).

Durch die Reified-Beziehung ist auch gesichert, dass es keine NM sondern eine 1:2 Beziehung ist, deshalb kann der Reifeid-Eintrag zwei konstante Referenzen haben.

Dazu wird statt des normalen Mapping Dictionary-Eintrags ein SymmReified-Eintrag verwendet, dieser verweist mit der Tabelle direkt auf die gesuchte Tabelle anstatt auf die NM Tabelle. Mit einem solchen Eintrag muss für die Anfragekonvertierung lediglich bestimmt werden welche SPARQL-Var an die erste Spalte und welche an die zweite Spalte gebunden wird, was durch die Symmetrie beliebig ist, allerdings konstant festgelegt werden muss.

4. Query Converter

4.1. Ansatz

Der Query Converter ist die Fortsetzung zu RDF2SQL (s. auf Seite 20). Zwar wäre es natürlich auch möglich, direkt SQL-Anfragen an die von RDF2SQL erstellte Datenbank zu stellen, allerdings ist es nur konsequent, wenn zur Datenbankkonvertierung auch noch eine Anfragenkonvertierung verfügbar ist. Hier kommt der Query Converter ins Spiel. Durch die Nutzung der Metadaten (s. Abb. , die während der Konvertierung der Datenbank gesammelt wurden, lässt sich, mit entsprechendem Aufwand, jede RDF-Anfrage in eine entsprechende SQL-Anfrage übersetzen.

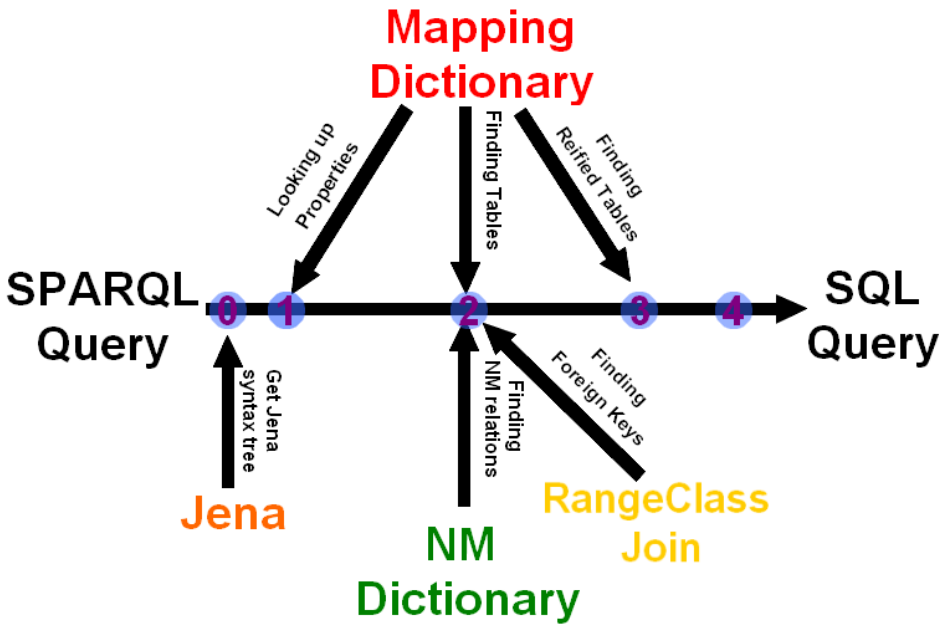

Abbildung 4.1.: Übersicht der Konvertierungsschritte und der genutzten Resourcen
0. Der Jena-Syntaxbaum wird in einen Querykonverter ***Syntaxbaum*** *umgewandelt und die entsprechenden* Nodes *erstellt.*
1. Die **Query Converter-*Variablen*** *werden erstellt und die Klassen, Properties und Zuverlässigkeiten werden für jeden* Node *bestimmt.*
2. Für jeden Node *werden die benötigten* **Selects, Tables *und* Conditions** *bestimmt.*
3. Behandlung der ***Sonderfälle****, wie Inverse und symmetrische reifizierte Tabellen.*
4. ***Vereinfachung*** *der SQL Anfrage.*

4.1.1. Einführende Beispielanfrage

An dieser Stelle ein oberflächliches Beispiel (Anfrage in Abb.4.2) um die Arbeitsschritte des Query Converters etwas zu illustrieren. Für weitere und ausführlichere Beispiele s. auf Seite 63. Im Verlauf dieses Kapitels wird auf die einzelnen Verfahren noch detaillierter eingegangen.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?xn ?cn ?ln
WHERE {
    ?x :name ?xn.
    ?x :capitel ?c.
    ?c :name ?cn .
    ?c :population ?p.
    Optional{
       ?c :locatedAt ?l .
       ?l :name ?ln
    }
    Filter (?p > 1000000)
}
```

Abbildung 4.2.: Einführende Beispielanfrage

Bevor die Anfrage bearbeitet werden kann muss der Syntaxbaum erstellt werden, diese Aufgabe ist an Jena ausgelagert und der Syntaxbaum wird im Jena-Programmverlauf abgefangen bevor Jena versucht die Anfrage an ein RDF Model zu stellen.

Es folgen nun die vier Konvertierungsschritte aus Abb.4.1. Es müssen die Informationen zu jeder CV gesammelt werden, durch das Optional gibt es zwei BGPs deren CVs getrennt betrachtet werden müssen. CV aus dem rechten Kind eines Optional-Knotens müssen als *unsafe* betrachtet werden, d.h. es kann nicht davon ausgegangen werden, dass die CV diese Property auch tatsächlich besitzt. Dies bedeutet, dass für die Klassenbestimmung einer CV diese Property nur innerhalb des jeweiligen Knotens genutzt werden kann, außerhalb nicht.

CV	Klasse	Known Properties	is Safe	Type
x	Country,Province	name, capital	$\surd$	Resource
xn	Attr. von x	-	$\surd$	Literal
c	City	name, capitalof, [locatedAt (unsafe)]	$\surd$	Resource
cn	Attr. von c	-	$\surd$	Literal
p	Attr. von c	-	$\surd$	Literal
l	Sea, Lake, River	flowsthrough	$\times$	Resource
ln	Attr. von l	-	$\times$	Literal

Tabelle 4.1.: CV-Informationen

```
Projection1( x xn cn l ) (
    Filter2 (
        LeftJoin3
            BGP4 (
              Triple5( x name xn )
              Triple6( x capital c )
              Triple7( c name cn )
              Triple8( c population p )
            )
            BGP9 (
              Triple10( c locatedAt l )
              Triple11( l name ln )
            )
        )
    )
)
```

Abbildung 4.3.: Syntaxbaum des Beispiels

Mithilfe der BGPs können nun die Eigenschaften der CVs, wie in Tabelle4.1, aus den Tripel erstellt werden.

Als nächstes müssen die beiden BGPs herausfinden, wo sie welche Property finden. Hierfür wird das MD benutzt und falls nötig Joins über weitere Tabellen gebildet (s. auf Seite 38). Durch das Optional muss ein Left Outer Join erstellt werden und BGP4 auf die linke Seite und BGP9 auf die Rechte (s. auf Seite 36) des Joins gestellt werden. Außerdem haben sowohl x als auch l nicht nur eine Klasse, deswegen müssen beide CVs zunächst ein Union (rote Teile in Abb.4.4, s. auf Seite 39) über ihre Klassen bilden. Der Filter findet erst relativ spät seine Anwendung, die Bedingung wird an entsprechende Tabellennamen angepasst und in die SQL Syntax übertragen (s. auf Seite 36). Letztendlich erhalten wir die SQL Anfrage in Abb.4.4.

```
Select

    BGP4.City_cn as Result_cn,
    BGP4.x_URI as Result_x,
    BGP4.xn_name as Result_xn,
    BGP9.l_URI as Result_l

From (

    Select c_City.URI as City_c_URI,
        c_City.population as City_p,
        c_City.name as City_cn,
        x_Union.URI as x_URI,
        x_Union.name as xn_name
    From
        City c_City,
        ((Select capital capital, URI URI, name name from Country)
        Union
        (Select capital capital, URI URI, name name from Province)
        ) x_Union
    Where (x_Union.capital = c_City.URI)) BGP4
    Left Outer Join (
        Select
            c_City.URI as City_c_URI,
            l_Union.URI as l_URI
        From
            locatedAt c_locatedAt,
            City c_City,
            ((Select URI URI from Lake)
            Union(Select URI URI from Sea)
            Union(Select URI URI from River)) l_Union
        Where
            (c_City.URI = c_locatedAt.City)
            and (l_Union.URI = c_locatedAt.Water)
            ) BGP9
    On (BGP4.City_c_URI = BGP9.City_c_URI)
    Where (BGP4.City_p > 1000000)
```

Abbildung 4.4.: SQL Anfrage des Beispiels

4.2. Grundlagen der Konvertierungen

Wie in Abbildung 4.1 zu erkennen ist wird die Anfragekonvertierung in einem schrittweisen Ablauf mit jedem Schritt vervollständigt. Grundlage aller folgenden Anwendungen sind die Informationen, die aus dem Syntaxbaum der ursprüngliche Anfrage gezogen werden können. Dafür wird der von Jena erstellte Syntaxbaum zur leichteren Benutzung in eine eigene Variante umgewandelt. In diesem Kapitel werden alle Datenstrukturen des benutzten Syntaxbaumes erläutert, insbesondere welche Funktion sie für die Konvertierung haben.

4.2.1. Query Converter Variablen (CV)

Die *Query Converter* Variablen (CV; sollte es nicht explizit anders erwähnt sein bezieht sich das Wort Variable immer auf die Query Converter Variablen) stellen eine interne Repräsentation der SPARQL-Variablen dar und beziehen sich daher nicht auf eine spezifische Tabelle, sondern auf die Menge aller Klassen, die die an die SPARQL-Var gestellten Bedingungen erfüllen. Hierbei werden CVs in drei Unterkategorien unterteilt.

Konverter Variable: Dieser Variablentyp repräsentiert die SPARQL-Standardvariablen und Variablen dieses Typs kommen mindestens einmal als Subjekt eines Tripels vor. Dies impliziert, dass sie als Platzhalter für *URIs* dienen. Sie besitzen ein oder mehrere Attribute, die die Menge ihrer möglichen Klassen einschränken.

Literal: Hierbei handelt es sich um die Repräsentation jener SPARQL-Variablen, die nur ein SPARQL-Literal repräsentieren. Sie kommen nur als Objekt von Tripeln vor und sind den entsprechenden Subjekt-Variablen als Attribut zugeordnet. Die möglichen Klassen eines Attributs sind irrelevant, da sie nur als Platzhalter für eine Spalte von einer oder mehreren Tabellen verwendet werden. Ein Attribut kann zum Beispiel der Name einer Country sein, oder aber auch die Hauptstadt. Kommt das Attribut auch als Subjekt eines Tripels vor, so wird es zu einer normalen CV erhoben.

Synonyme: Hierbei handelt es sich um eine Hilfskonstruktion um das Fehlen der selbsterstellten Variablen außerhalb des Select-Blocks in *SQL* zu umgehen. In *SPARQL* ist es in der *HAVING* Klausel des *Group by's* (s. auf Seite 37) möglich Ausdrücke in einer Variable zu speichern und umzubenennen, wohingegen in *SQL* immer wieder der komplette Ausdruck verwendet werden muss. Synonym-Variablen sind in der Lage anstelle ihres Namens nun den kompletten Ausdruck in die SQL Anfrage zu schreiben.

Die CVs werden in Postorder erstellt, da nur die Blätter des Syntaxbaumes, die Triples, in der Lage sind Variablen zu erstellen (mit Ausnahme der Group by-Knoten, auch er kann Variablen erstellen, falls in dem *Group by*-Ausdruck eine Variable neu definiert wird). Die Klassenbestimmung wir mit Hilfe einer Liste der bekannten Properties (Prädikate die dieser CV zugeordnet sind), die jede CV speichert, durchgeführt. Ist diese Liste leer, so handelt es sich um ein Attribut. Wie später noch genauer erläutert wird, können nicht alle Properties einer CV für die Klassenbestimmung verwendet werden, falls diese als unsafe gelten (s. auf Seite 43). Die restlichen Properties werden in dem Mapping Dictionary gesucht und

ihre möglichen Klassen notiert. Die CV kann nur noch Objekte von Klassen annehmen, die bei der Schnittmenge aller möglichen Klassen für jede Propertys übrig bleiben.

4.2.2. Allgemeines zu Knoten

Um die Umwandlung von SPARQL Anfragen zu SQL Anfragen vollziehen zu können ist es notwendig, dass jeder Knoten folgende Anforderungen erfüllt:

- **Die Knoten sind in der Struktur des Syntaxbaumes angeordnet.**

  Für sich alleine und ohne Kontext ist jeder Knoten nutzlos, die kleinste Anfrage besteht schon mindestens aus 3 Knoten: einem Tripel, einem BGP und einer Projektion.

- **Informationen von gleichen CVs zusammenführen.**

  Wenn ein Knoten von seinen Kindern die gleichen CVs erhält, sie aber unterschiedliche Informationen enthalten, so muss der Knoten eine CV erstellen, die diese Informationen sinnvoll miteinander verbindet.

- **Knoten wissen welche CVs bei ihm und allen seinen Nachfolgern vorkommen und welche Properties diese bis zu diesem Punkt haben.**

  Das ist vor allem notwendig um die Klassen richtig zu bestimmen und nicht zu früh die Auswahl einzuschränken.

- **Knoten können feststellen welche CVs sie benötigen**

  Da die selektierten Spalten in der SQL-Anfrage umbenannt werden müssen, muss ein Knoten feststellen können, welche CVs er benötigt, damit diese in der SQL Anfrage richtig selektiert werden können.

- **Knoten können ihre Kinder nach einem CV fragen und bekommen entweder den SQL Variablennamen der der CV innerhalb des Kindes entspricht oder die Mitteilung, dass dieses Kind die CV nicht kennt.**

  Da nicht in allen Kindern eines Knotens alle CVs vorkommen, die für den Vater-Knoten benutzt werden, ist es notwendig, dass die höheren Knoten ihren Kindern mitteilen, welche CVs sie selbst benötigen. Durch diese Informationen können dann die Namen der benötigten Spalten nach oben mit gezogen werden, bis zu dem Punkt an dem sie verwendet werden.

4.2.3. Tripel

In dem zur Anfrage zugehörigen Syntaxbaum bilden die *Tripel* die Blätter des Baumes. Obwohl sie die grundlegendsten Informationen der Anfrage beinhalten, geben sie einzeln betrachtet leider nur sehr vage Informationen über die Variablen. Für die Auswertung der Anfrage sind die Beziehungen, die die *Tripel* zwischen den Variablen aufbauen umso wichtiger. Jeder Tripelknoten speichert daher nur die Information über die Beziehung seiner enthaltenen Variablen, damit diese dann wie im nächsten Kapitel beschrieben ausgewertet werden können. Eine Ausnahme bilden Tripel der Form *?Variable a Klasse*, deren einzige Aussage ist, dass die Subjekt-Variable unbedingt nur die angegebene Klasse annimmt.

Bei der Konvertierung in die SQL-Abfrage werden die Properties der Tripel aber nicht nur zur Klassenbestimmung der CVs benutzt. Mit ihnen wird auch die Spalte der entsprechenden Tabellen in der SQL-Datenbank angesprochen, die die benötigten Daten enthält. Um diese Spalte zu identifizieren, wird die bereits erwähnte Meta-Tabelle *Mapping Dictionary* (MD) benötigt. Schließlich kann man nicht davon ausgehen, dass die Spalte der Tabelle den gleichen Namen wie die Property besitzen muss. Gleichermaßen können die Spalten mehrerer Tabellen unterschiedlich benannt sein, auch wenn diese die Informationen der selben Property speichern. Daher wird auch ein Schlüsselpaar aus Klasse und Property benötigt, um die gesuchte Tabellen und Spalte eindeutig zu finden.

4.2.4. BGP - Basic Graph Pattern

Das *BGP* im Syntaxbaum von *SPARQL* stellt eine Zusammenfassung der ihm zugehörigen *Tripeln* dar. Durch die Informationen (Subjekt - Prädikat - Objekt), die in den *Tripeln* enthalten sind, lässt sich mit einem *BGP* ein Graph aufbauen, indem man die Subjekte und Objekte als Knoten betrachtet und die dazugehörigen Prädikate (Properties) als Kanten zwischen diese zieht. Aufgrund dieser Struktur ist es schon mit einem *BGP* möglich erweiterte Eigenschaften der Anfrage auszuwerten.

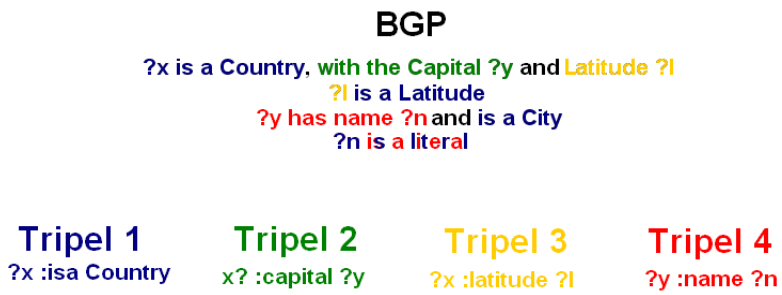

Abbildung 4.5.: Informationssammeln eines BGPs: Die Farbe stehen jeweils für das entsprechende Tripel. Es ist zu erkennen, dass durch mehr Informationen zusätzliche Schlussfolgerungen insbesondere über die Klassen der CVs getroffen werden können. So könnte beispielsweise das Tripel 4 keinerlei Aussage über die Klasse von ?y machen und es wäre auch nicht klar ob ?n ein Literal oder eine Entität wäre.

Es sind zum einen logische Schlüsse über den Zusammenhang von Variablen durch die Subjekt-Objekt-Verbindung möglich. Enthält zum Beispiel ein Tripel die Informationen *?x Prädikat1 ?y* und ein weiteres die Information *?y Prädikat2 ?z*, so kann man durch die einzelne Betrachtung des ersten Tripels nur für die Konvertierung in Erfahrung bringen, dass die entsprechende Spalte des *Prädikat1* für die Hometables von *?x* in der Variablen *?y* gespeichert wird (entsprechend für das zweite Tripel). Führt man aber die Informationen beider Tripel zusammen erkennt man, dass die Variable *?y* nicht nur zur Identifizierung dieser Spalte dient (ein Attribut von *?x* ist), sondern selber auf eine oder mehrere Tabellen verweist, die mit den Tabellen von *?x* über die Spalte von *Prädikat1* gejoined werden müssen. Dies entspricht also dem Join über einen Fremdschlüssel in *SQL*.

Zum anderen werden die möglichen Klassen einer Variablen durch ihre Prädikate eingeschränkt. Könnte eine Variable *?x* ohne weitere Informationen alle möglichen Klassen enthalten, so kann diese durch ein Tripel wie im Beispiel oben schon erheblich eingeschränkt

werden. Durch die Information *?x Prädikat1* kann man nun die Klassen von *?x* auf alle Klassen beschränken, die die Eigenschaft *Prädikat1* besitzen. Dies kann ganz einfach im *MappingDictionary* nachgeschlagen werden. Fügt man nun mehrere Tripel dieser Form in einem *BGP* zusammen so wächst natürlich der Informationsgewinn. Schlägt man für jedes Tripel wie beschrieben die Klassen von *?x* nach, so kann diese nach der Zusammenfügung nur noch die Schnittmenge der einzelnen Klassenmengen haben. Dieser Schritt ist essentiell für die Überführung in eine SQL-Abfrage, da die möglichen Klassen einer Variablen den Tabellen in der Datenbank entsprechen, die die benötigten Datensätze enthalten. Hat eine Variable mehr als eine mögliche Klasse, so muss also ein UNION-Unterabfrage für diese CV gebildet werden. Dieser Vorgang wird im Kapitel Union/Multi-KlassenVariablen (s. auf Seite 39) genauer erläutert.

Des Weiteren werden durch die einzelnen *BGPs* die Informationen der Anfrage gruppiert. So wird zum Beispiel nur ein *BGP* von einem Filter beeinflusst, der dann später mit anderen *BGPs* zusammengeführt wird. So können für die Optimierung Unterabfragen zuerst gefiltert werden, anstatt sonst die größere gejointe Datenmenge zu filtern. Die Gruppierung von Informationsblöcken führt natürlich auch dazu, dass diese in übergeordneten Knoten wieder zusammengeführt werden. Hierfür wird mit Ausnahme des Optional-Knotens eine einfache Mengenvereinung durchgeführt.

Ab dem BGP können Knoten als klassische SFW-Blöcke in SQL dargestellt werden. Ein BGP bildet damit also im Allgemeinen die am tiefsten liegenste Unterabfrage eines Zweigs der Anfrage. Die Informationen der einzelnen Tripel werden ausgewertet und die Beziehungen unter den einzelnen CVs auf Joins überprüft. Die passenden Tabellen zu den Klassen einer CV werden in den From-Abschnitt gesetzt und mit der jeweiligen CV markiert, damit mehrmals die selbe Tabelle für unterschiedliche CVs benutzt werden kann. Für die identifizierten Joins werden die benötigten Join-Bedingungen erstellt und dem Where-Abschnitt hinzugefügt. Der Select-Abschnitt wird noch nicht durch das BGP gefüllt, da erst durch die übergeordneten Knoten bestimmt wird, was von ihren Kindern benötigt wird.

4.2.5. Projection

Der Projection-Knoten bildet in jedem Syntaxbaum die Wurzel. Er speichert, welche CVs in der Ergebnisausgabe benötigt werden. Wie oben erwähnt bestimmen die übergeordneten Knoten, was im Select-Abschnitt der Kinder benötigt wird. Da der Projection-Knoten die Wurzel ist, beginnt jede Rekursion bei diesem Knoten, allerdings sind alle Rekursionen so aufgebaut, dass sie erst auf dem Rückweg tatsächliche Operationen durchführen und somit wird der Projection-Knoten immer als letztes berechnet. Die Anfragen gehen rekursiv bis zu den Blättern des Baumes bis ein Knoten gefunden wurde, in dem Informationen zu dieser CV gespeichert sind. Dies ist natürlich ein Tripel, in dem die CV vorkommt, doch o.B.d.A. kann man sagen, dass die Suche in einem BGP endet, da erst hier die gesuchte Tabellenspalte in dem Select-Abschnitt eingetragen werden kann. Die Spalte wird aus den Unterabfragen bis zu dem SFW-Blocks des Projection-Knotens geleitet, welche diese als Ergebnisspalte ausgibt.

4.2.6. Filter

Der Filter ist einer der einfachsten, aber doch wichtigsten Knoten nach dem BGP, da er Filterbedingungen beinhalten kann. Dieser Knoten besitzt nur ein Kind auf das die Filterbedingungen wirkt.

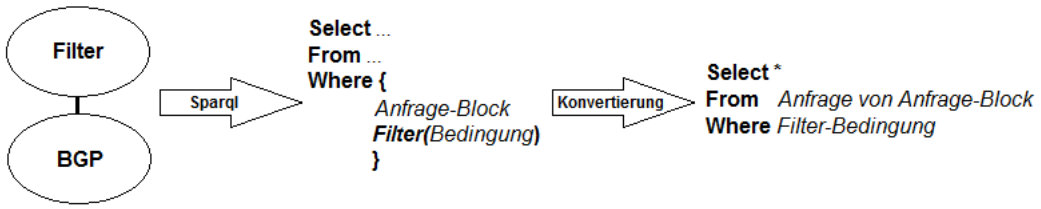

Abbildung 4.6.: Konvertierung von Filter-Knoten

Abgesehen von den Namen *Filter* und *Where* unterscheiden sich die Klauseln nur in Punkten der Syntax. Ein Filterknoten muss lediglich die beinhalteten CVs in den Bedingungen in die SQL Variablennamen umändern und fügt die fertige Bedingung dem Where-Abschnitt hinzu. Da Filter eigenständige Knoten sind, bilden sie rein theoretisch einen eigenen Abschnitt in der SQL-Anfrage. Im Falle, dass der Filter-Knoten als Kind eines Group By-Knoten besitzt wird er mit diesem kombiniert, da er aus einem Having anstatt eines Filter-Abschnitts entstanden ist.

4.2.7. Optional

Dieser Knoten führt immer zu einem neuen Zweig im Syntaxbaum und hat daher zwei Kinder. Das linke Kind spiegelt dabei den äußere Teil der Anfrage wieder, während das rechte Kind den Informationsblock innerhalb des „Optional“-Operators enthält. Jegliche Informationen und Einschränkungen, die dieses Kind enthält, sind, wie der Name schon sagt, optional. Aus diesem Grund ist die Reihenfolge im Syntaxbaum nicht irrelevant. Der Umstand, dass die Informationen dieses Kindes, zum Beispiel zusätzliche Properties einer Variable, nicht unbedingt erfüllt sein müssen, führt dazu, dass diese Informationen zur globalen Klassenbestimmung, wie oben beschrieben, außerhalb des Kindes nicht verwendet werden dürfen. Diese Properties werden also als „unsafe“ markiert, damit sie nach der Informationsvereinigung in den übergeordneten Knoten identifiziert werden können. Innerhalb des Kindes müssen diese Informationen aber beachtet werden, wodurch zusätzlich zu der globalen Informationsstruktur auch eine lokale, Knoten-spezifische Speicherung der bis dorthin bekannten Informationen notwendig wird.

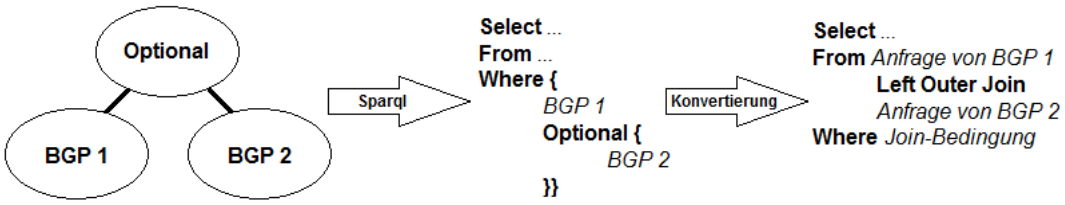

Abbildung 4.7.: Konvertierung von Optional-Knoten

Die Optionalität lässt sich in SQL als *Left Outer Join* umsetzen. Als Erinnerung: Im Gegensatz zu dem normalen *Join* bleibt hier die linke Tabelle vollständig und lediglich die Werte der rechten Tabelle, die die *Join Condition* erfüllen, werden an die Werte von Links gejoint. Das linke Kind und rechte Kind werden also als Unterabfragen im From-Abschnitt behandelt und über ein *Left Outer Join* zusammen gefügt. Hierbei treten aber einige Besonderheiten für die Konvertierung auf, die auf die Unterschiede zwischen SQL und SPARQL zurückzuführen sind, wie zum Beispiel den Join über Null-Werte (s. auf Seite 43).

4.2.8. Group by

Das Group by ermöglicht sowohl in SQL als auch in SPARQL das Verwenden von Aggregationen. In beiden Anfragesprachen sind die Group by-Funktionen außerhalb des SFW-Blocks, was bedeutet, dass diese ein wenig mehr Sonderbehandlung brauchen als andere Knoten.

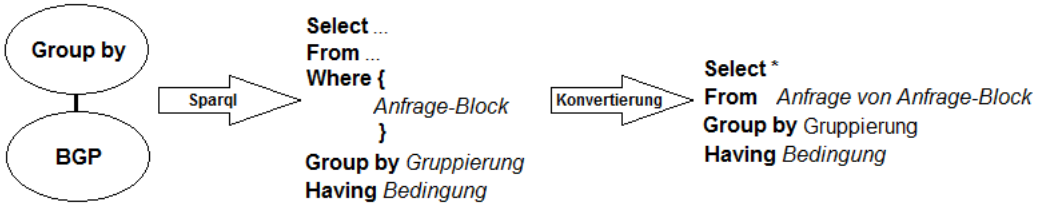

Abbildung 4.8.: Konvertierung von Group by-Knoten

Außerdem sind sie neben den Tripeln die einzigen Knoten, die in der Lange sind CVs zu erzeugen. Diese CVs stellen SQL vor das Problem das es ohne PSQL (oder ähnliches) nicht in der Lage ist dieser Stelle neue Variablen zu erstellen und diesen einen Wert zuzuweisen, dies wäre nur im Select-Block möglich. So kann man beispielsweise nicht eine Variable einführen die den Wert von pi als Konstante, oder die Summe von zwei anderen Variablen zugewiesen bekommt. In SPARQL ist letzteres bei den Aggregationen möglich. *Group by (?x + ?y as ?z)* würde das Ergebnis in der neuen CV *?z* speichern. Hier kommt die dritte Klasse von CVs ins Spiel, die Synonyme (s. auf Seite 32). Sie werden intern als CVs behandelt, speichern aber den gesamten Ausdruck. So würde *?z* intern als einzelne Variable behandelt, aber extern in der SQL Anfrage würde wieder *?x + ?y* stehen. Aus diesem müssen diese CVs und auch Filter gesondert und nur innerhalb des Group by-Knotens behandelt werden, da sie außerhalb des Knotens den Namen ändern können und das Synonym nicht die beteiligten CVs und Operatoren, sondern schlicht den gesamten Ausdruck speichert. Trotzdem ist das Group by uneingeschränkt nutzbar, abgesehen von Aggregationsfunktionen im Select, da diese leider durch die Begrenzung der verwendeten Jena-Version nur in der Having Klausel verwendbar sind. Da wir aber grundlegend auf den Syntaxbaum von Jena angewiesen sind, können wir dieses Problem auch nicht umgehen.

Auch bei der Optimierung nimmt das Group by eine besondere Position ein. Obwohl es nur ein Kind hat, kann es wegen seiner gesonderten Filter und CVs nicht mit diesem zusammen gefasst werden. Dies würde aber auch syntaktisch in SQL nicht korrekt sein, von daher ist dies kein Verlust, aber es stellt eine Unregelmäßigkeit in der Programmstruktur da.

Im Beispiel in Abbildung 4.9 sehen wir ein Beispiel für ein Group by. Ziel dieser Anfrage ist es alle Länderkürzel von Ländern auszugeben, deren durchschnittliche Bevölkerung in den Städten über 100000 Einwohner liegt.

———-SPARQL Anfrage———

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?cc
WHERE {

    ?cou a :Country .
    ?cou :carCode ?cc .
    ?cou :hasCity ?x .
    ?x :population ?xp. }

Group by ?cc
Having ((sum(?xp)/count(?x)) > 100000)
```

———-SQL Anfrage———

```
Select Country_cc as Country_cc
From (

    Select
        x_City.population as City_xp,
        x_City.URI as City_x_URI,
        cou_Country.carCode as Country_cc
    From
        cityIn cou_cityIn,
        Country cou_Country,
        City x_City
    Where
        (cou_Country.URI = cou_cityIn.AdministrativeArea)
        and (x_City.URI = cou_cityIn.City)
    )

GROUP BY Country_cc
HAVING (( Sum(City_xp) / Count(City_x_URI)) > 100000)
```

Abbildung 4.9.: test-complex-groupby.sparql

4.3. Besonderheiten bei der Konvertierung

4.3.1. Joins

Zur Identifikation der benötigten Tabellenspalte für eine Property wird diese in dem Mapping Dictionary (s. auf Seite 20) für jede Klasse der CV nachgeschlagen. Falls die Property

einer CV nicht in ihrer Hometables zu finden ist, so muss es sich ein einer NM-Tabelle befinden. Eine Liste aller NM-Tabellen lässt sich in der Meta-Tabelle NMTab finden, mit der damit überprüft werden kann zu welchem Entitätstypen eine Tabelle gehört.

Nun werden aber nicht alle Einträge dieser Spalte der NM-Tabelle gesucht, sondern nur Einträge, die in Relation mit den Einträgen der Hometables der CV stehen. Die Filterung wird mit einem Join umgesetzt, der die URI-Spalte der Hometables mit der NM-Tabelle verknüpft. Damit herausgefunden werden kann, welche Spalte der NM-Tabelle für den Join gedacht ist, muss die NMJ (kurz für NM-Join) herangezogen werden. In dieser kann über die Klasse der CV und das gesuchte LookupAttr (Property), die Spalte für den Join nachgeschlagen werden.

Diese Überprüfung findet im BGP statt und die Join-Bedingung wird gleich im Where-Abschnitt eingetragen.

4.3.2. Union/Multi-Klassen Variablen

Wie bereits mehrmals erwähnt kann eine CV viele verschiedene Klassen abdecken. Jede Klasse verweist auf eine oder im Falle von Oberklassen auf mehrere Tabellen in SQL. Objekte der Klasse „Country“ sind so zum Beispiel in der Tabelle „Country“ zu finden, wohingegen Objekte der Klasse „ANY“ in allen Tabellen stehen und Objekte der Klasse „Water“ in den Tabellen „River“, „Sea“ und „Lakes“.

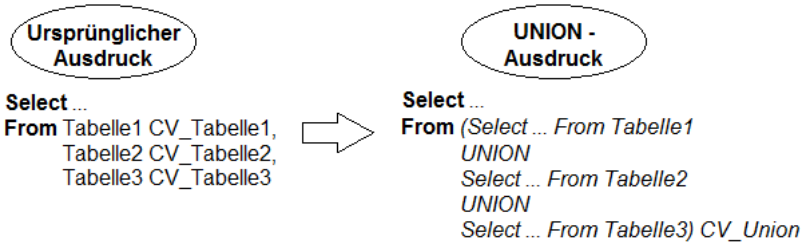

Abbildung 4.10.: Union-Konvertierung

Damit eine SQL-Anfrage nun aber auch Multiklassen Variablen wie in RDF verwenden kann, muss ein Union über alle benötigten Tabellen gebildet werden. Um herausfinden zu können welche Tabellen dafür infrage kommen, wird eine einfache Klassenbestimmung für die Variable durchgeführt. Falls mehr als eine Tabelle für die selbe Variable benötigt wird, wird über alle ein Union-Ausdruck gebildet, der als kleine Unterabfrage im From-Abschnitt die einzelnen Tabellen ersetzt. Bei einem Unions-Ausdrucks wird immer sichergestellt, dass durch Definition der Klassenbestimmung in der Union-Unterabfrage alle benötigten Spalten auch in allen Tabellen vorkommen müssen. Sucht man für eine Klasse eine bestimmte Property, die die anderen Klassen nicht besitzen, so muss ein Optional-Ausdruck hierfür verwendet werden. Die Union-Findung wird schon im BGP ausgeführt, damit darauf folgende Knoten das Union wie jede andere Variable auch behandeln können, in Abb.4.11 zu erkennen ist. Zunächst wird erst einmal der Union-Block erstellt (rot gefärbter Teil) und gekapselt und dann von den höheren Knoten wird dieser wie eine gewöhnliche Tabelle verwendet (blau gefärbter Teil) . Dies spielt gerade eine wichtige Rolle für die Übergabe der Tabellen- und Spaltennamen an übergeordnete Knoten, die diese an-

gefragt haben.

————-SPARQL Anfrage————

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?xn
WHERE  {
    ?x :name ?xn.
    }
```

——-SQL Query——-

```
Select BGP3.xn_name as Result_xn,
From (
    Select x_Union.name as xn_name
    From (
    (Select URI URI, name name from Country)
       Union(Select URI URI, name name from City)
       Union(Select URI URI, name name from Religion)
       Union(Select URI URI, name name from Desert)
       Union(Select URI URI, name name from Language)
       Union(Select URI URI, name name from Island)
       Union(Select URI URI, name name from Organization)
       Union(Select URI URI, name name from Mountain)
       Union(Select URI URI, name name from Continent)
       Union(Select URI URI, name name from Lake)
       Union(Select URI URI, name name from Sea)
       Union(Select URI URI, name name from EthnicGroup)
       Union(Select URI URI, name name from Islands)
       Union(Select URI URI, name name from Mountains)
       Union(Select URI URI, name name from Province)
       Union(Select URI URI, name name from River)
       ) x_Union
    ) BGP3
```

Abbildung 4.11.: test-optional.sparql

4.3.3. Anfragen an Reified - Tables

Wie bereits auf Seite 19 beschrieben gibt es Beziehungen, die über eine n:m Beziehung hinausgehen. Auch in der Konvertierung muss zwischen den SymmReified und den gewöhnlichen Reified unterschieden werden.

In Abb. 4.12wird eine gewöhnliche Reified-Beziehung anhand der isMember-Property vorgestellt. In der Abfrage wird die Relation zwischen Ländern und Organisationen, für

die zusätzlich der Typ der Mitgliedschaft gespeichert ist, erfragt. Die reifizierte Property zu isMember heißt Membership mit den Properties ofMember, inOrganization und type. Wie die Abbildung zeigt, wird ein normaler Join wie über eine NM-Tabelle mit den CV ?x und ?o ausgeführt.

———-SPARQL Anfrage———

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?xn ?on ?t
WHERE {

    ?x a :Country.
    ?x :name ?xn .
    ?m a :Membership.
    ?m :type ?t.
    ?m :ofMember ?x.
    ?m :inOrganization ?o .
    ?o :name ?on

}
```

———-SQL Anfrage———

```
Select

    x_Country.name as Result_xn,
    o_Organization.name as Result_on,
    m_Membership.type as Result_t

From

    Membership m_Membership,
    Country x_Country,
    Organization o_Organization

Where

    (m_Membership.ofMember = x_Country.URI)
    and (m_Membership.inOrganization = o_Organization.URI)
```

Abbildung 4.12.: Anfrage an eine reifizierte Tabelle (test-isMember.sparql)

Bei Symmreified-Beziehungen verweisen, anders als bei normalen Reified-Tabellen , beide Seiten der Relation auf die selbe Tabelle, wodurch sie die selbe Property erhalten. Damit stellt sich als erstes das Problem, diese Klassen von den konventionellen reifizierten Klassen zu unterscheiden. Dadurch dass mit der selben Property der reifizierten Klasse zwei unterschiedliche Spalten der selben Tabelle angesprochen werden, werden auch zwei Mapping Dictionary-Eintrage durch RDF2SQL erstellt. Da mehrere Mapping Dictionary-Einträge bei einer einzelnen Anfrage mit (Klasse, Property) lediglich bei SymmReified-Spalten vorkommen, kann dies in der Auswertung als Entscheidungskriterium benutzt

werden (s. auf Seite 26). Dies schafft aber auch das Problem, dass bei mehrfacher Verwendung der Property entschieden werden muss, welche CV an welche Spalte gebunden wird. Die Abb. 4.13 zeigt diese Probematik bei der bordering-Property. Die reifizierte Klasse von bordering ist Border, über die mit der bordering-Property zwei Länder verbunden werden und zusätzlich die Länge dieser Grenze erreichbar ist. Die SQL-Tabelle Border enthält die Spalten URI, Country1, Country2 und Length, wobei die URI-Spalte zu vernachlässigen ist, da reifizierte Klassen in der Regel keine spezifisch benannte URI haben, sondern nur eine interne fortlaufende Bezeichnung. In diesem Beispiel muss entschieden werden, ob die CV ?x oder ?y an die Country1-Spalte gebunden wird. Ebenfalls muss später entsprechend darauf reagiert werden und die jeweils andere an Country2 gebunden werden.

———-SPARQL Anfrage———

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?y ?b
WHERE {

    ?b a :Border.
    ?b :bordering ?y .
    ?b :bordering ?x .

}
```

——-SQL Query——-

```
Select

    b_Border.Country2 as Result_x,
    b_Border.Country1 as Result_y,
    b_Border.URI as Result_b

From

    Border b_Border
```

Abbildung 4.13.: Anfrage an eine symmetrische reifizierte Tabelle (test-onlyreified.sparql)

4.3.4. Geschachtelte Anfragen

Geschachtelte Anfragen stellen keine besonderes Hindernis da, weil alle Knoten so aufgebaut sind, das sie kontextfrei funktionieren, d.h. es spielt für sie keine Rolle, welche Art von Knoten über oder unter ihnen liegen. Alle benötigten Informationen werden selbst gewonnen oder über die Superklasse *Node* bezogen.

Eine Ausnahme bilden hier die *BGP*s, sie müssen unterscheiden ob es sich bei ihren Kindern um Tripel handelt oder um einen anderen Knotentypen, da kein Requestinformation auf Tripel möglich ist, da sie nicht genügend Wissen über die CV's haben.

4.3.5. Inverse Beziehungen

In RDF-Modellen sind durchaus beidseitige Beziehungen zwischen zwei Entitäten üblich. In SQL sind diese Beziehungen nur in eine Richtung gespeichert, um keine unnötige Redundanz einzuführen. Um in Anfragen trotzdem inverse Properties verwenden zu können gibt es im Mapping Dictionary eine Markierung für solche. Für diese kann dann in der Inversetable-Tabelle die andere Richtung der Beziehungen gefunden werden. Auf Properties die nicht im Mapping Dictionary verzeichnet sind, reagiert der Query Converter damit, diese zunächst in der Inversetable zu suchen. Wird der Konverter fündig so stellt er die Anfrage um, so dass das inverse Attribut verwendet wird und kann die korrekte Anfrage generieren. Wurde wie in Abb.4.3.5 beispielsweise nach River-Einträgen gefragt, die über die flowsThrough Property mit Lakes verbunden ist. Diese Property ist laut MD ein inverses Attribut und es wird daher für den Join die andere Richtung von Lake nach River über die Property flowsThrough_Inv gewählt.

———-SPARQL Anfrage———

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?RN ?LN
WHERE {

    ?R a :River;

        :name ?RN;
        :flowsThrough ?L .

    ?L :name ?LN }
```

——-SQL Query——-

```
Select

    R_River.name as Result_RN,
    L_Lake.name as Result_LN

From

    River R_River, Lake L_Lake

Where

    (L_Lake.flowsThrough_Inv = R_River.URI)
```

Abbildung 4.14.: test-flowsThrough.sparql

4.3.6. Joins mit Optionals und Nullwerten

Beim Join von BGPs über zwei Optional-Knoten tritt ein besonderer Unterschied zwischen SPARQL und SQL zum Vorschein. Entscheidend ist das (s. auf Seite 17) schon angesprochene Verhalten beim Join über Null-Werte. Für CV, die mehrmals optional gebunden werden, müssen daher drei Fälle betrachtet werden:

1. Beide Tripel mit der CV binden ein Ergebnis
2. Beide Tripel binden kein Ergebnis (Null-Wert)
3. Nur einer der beiden Tripel bindet ein Ergebnis

Das Verhalten von SPARQL/RDF in diesem Fall sieht dabei vor, dass alle drei Fälle in die Ergebnismenge übernommen werden. SQL würde zum Vergleich nur den ersten Fall akzeptieren. Bei diesem müssen aber sowohl in SQL als auch in SPARQL die Werte gleich sein. Der dritte Fall erscheint im ersten Moment etwas merkwürdig, ergibt aber unter näherer Betrachtung der Baumstruktur etwas mehr Sinn. So kann die CV entweder über den einen Teilbaum an einen Wert gebunden werden oder durch den Anderen.

SQL behandelt diese Fälle anders. Es joint Null nicht an Nicht-Nullwerte. Es ist zwar möglich Nullwerte durch eine Funktion mit einem Wert zu ersetzen, der als normaler VarChar2 oder ähnliches behandelt wird, aber diese würden nur dann normal gejoint werden, wenn beide Zweige die selbe interne Nullwert-Ersetzung hätten. Dies würde bedeuten, dass beide Werte gleich sind, was genau der Fall ist, den wir nicht haben wollen.

Um nun das Verhalten von SPARQL/RDF in einer SQL-Anfrage zu erhalten, muss man also einen anderen Weg nutzen. Der normale equi-Join muss durch die expliziten Bedingungen erweitert werden, dass die einzelnen Teilbäume einen Null-Wert liefern können. Die passende Umformung für diesen Effekt ist:

$(x.value = y.value) \vee (x.value\ is\ null \wedge y.value\ is\ not\ null)$

$\vee(x.value\ is\ not\ null \wedge y.value\ is\ null) \vee (x.value\ is\ null \wedge y.value\ is\ null)$

Bei SQL-Anfragen kommt zudem erschwerend noch dazu, dass die beiden Spalten in einer Selektion zusammengefasst werden muss. Bei einem normalen equi-Join ist dies kein Problem, da beide Spalten den selben Wert aufweisen müssen. Nun muss aber die jeweils andere Spalte selektiert werden, wenn eine der beiden Spalten den Null-Wert liefert. Dies ist mit einer Case-Bedingung im Select-Abschnitt möglich, die folgende Struktur hat:

$CASE\ WHEN\ x.value\ is\ null\ THEN\ y.value\ ELSE\ x.value\ END$

Der gesamte Komplex ist noch einmal in Abb. auf dieser Seite abgebildet.

```
Select Case When x.value is null then y.value else x.value END
[...]
On (
    (x.value = y.value) or
    (x.value is null and y.value is not null) or
    (x.value is not null and y.value is null) or
    (x.value is null and y.value is null)
)
```

Abbildung 4.15.: Join mit Optionals

5. Entwurf und Implementierung

5.1. Programmablauf

Der Query Converter nutzt die Daten und Metadaten von RDF2SQL um damit eine SPARQL-Anfrage in eine äquivalente SQL-Anfrage auf eine von RDF2SQL erstellte Datenbank um zu wandeln und auszuführen.

Die Query-Konvertierung kann in 5 Schritte unterteilt werden. Jeder Schritt wird auf den Wurzelknoten ausgeführt, der dann rekursiv die Funktion auch bei den Folgeknoten aufruft.

1. Eine gegebene Query *q* wird als erstes mit der Algebra-Funktion aus dem ARQ Paket von Jena[7] in ein Op-Objekt umgewandelt, mit dem der TreeBuilder gestartet wird. Das Op-Objekt liefert den Syntaxtree mit dem die Knoten des Baumes für den Query Converter erstellt werden. Der Treebuilder führt die Umwandlung unter Zuhilfenahme des Treewalkers und eines TreeVisitors durch. Diese werten den Algebraausdruck Knoten für Knoten aus und erstellen die entsprechenden QC Nodes. Auf dem Baum aus QC Nodes wird dann letztlich gearbeitet. Der QC-Syntaxbaum besteht ausschließlich aus Objekten die von der Hauptknotenklasse Node geerbt haben. An der Spitze jedes Syntaxbaumes ist immer eine Projektion.

2. Als nächstes werden die *Query Converter*-Variablen (CV)(s. auf Seite 32) angelegt. Die CV haben ihren Ursprung an den Tripeln, werden aber erst in den BGPs in einer lokalen Liste gespeichert. Die Variablen werden dann mit ihren bekannten Attributen an die Elternknoten weitergeleitet, die dann die Informationen von ihren Kindern zusammenführen. An der Wurzel sind somit also alle CVs mit allen Attributen bekannt, also so viel über die *SPARQL* Variablen bekannt wie möglich. Dieses Wissen wird nun mit allen Kindern geteilt und separat gespeichert, so dass jeder Knoten Zugriff auf die gesamten CVs hat. Attribute aus der rechten Seite eines Left Outer Joins werden bei Knoten oberhalb des Left Outer Joins allerdings als "unsafe" markiert, weil Attribute aus einem Optional-Abschnitt nicht für die Bestimmung der Klasse genutzt werden können, da sie nicht zwangsläufig zutreffen müssen.

3. Danach werden die Informationen (CVs) für jeden Knoten, die er für sich selber benötigt, gesammelt. Diese werden an die Kinder weitergegeben, die prüfen ob sie lokal Informationen über die CVs besitzen. Ist dies nicht der Fall, leiten sie wiederum die Anfrage rekursiv weiter bis die Rekursion auf ein Group by (s. *auf Seite 37*) (das nur noch die gegroupten Variablen und Aggregatfunktionen zur Verfügung stellt) oder Triple trifft. In der Rückrichtung der Rekursion werden nun die Ergebnisse der Kinder in InformationSets zusammengeführt und die jeweils benötigten Selektierungen gespeichert und nach oben weitergegeben. Neben der Selektierung werden auch Bedingungen bestimmt, wie z.B. in einem *BGP* das Verbinden von NM-Tabellen durch

Joins oder das Anfügen der Bedingungen eines *FILTER*s (s. auf Seite 36). Durch diese Bedingungen können weitere benötigte Informationen (Tabellenspalten zum Joinen etc.) auftreten, die wieder von den Kindern angefordert werden müssen. So baut sich die Anfrage von selbst und es werden nur Spalten selektiert, die auch später auch verwendet werden.

4. Dieser Schritt dient der Optimierung (s. auf Seite 59). Dabei werden größtenteils *InformationSets* von geschichteten Knoten zusammengefasst, sofern der Knoten nur ein Kind hat, damit die erstellte Anfrage nicht zuviele sperrige Unterabfragen benutzt. Ebenfalls werden gewisse Optiemierungen innerhalb der InformationSets durchgeführt, wie zum Beispiel die Behandlung von Chain Conditions (s. *auf Seite 59*) und die Entfernung von nicht mehr benötigten Tabellen (s. auf Seite 59).

5. Im letzten Schritt wird nun aus den gesammelten Informationen des Syntaxbaumes rekursiv die SQL Anfrage gebaut und anschließend an die Datenbank gestellt und das Ergebnis in dem geforderten Format ausgegeben.

5.2. Interne Strukturen

- Konvertervariable (CV)
  - PropertySet
- InformationSet
  - Select
  - From
  - Condition
- RequireSet
- Node
  - Tripel
  - Projection
  - BGP
  - Filter
  - Join
  - Left Outer Join
  - Group By

5.2.1. Query Converter Variable (CV)

Query Converter-Variablen werden für jeden Knoten von den Kindknoten kopiert, sie teilen sich also nicht die Variablen direkt, sondern nur die Inhalte. Grund dafür ist, dass Informationen wie safe nur lokal gelten und wenn sie auf die selben Variablen zeigen würden,

würde einige Knoten den anderen in ihren CV Veränderungen erzeugen. Knoten haben eine lokale Liste mit CVs namens CVList, die alle Informationen enthält, die soweit in dieser Stufe des Baumes vorhanden sind. So sind zum Beispiel nicht in diesem Teilbaum benutzte Variablen nicht in dieser Liste vorhanden. Zusätzlich besitzen die Knoten aber auch eine zweite Liste, die FullCVList, die die CVs des Wurzelknotens, also CVs mit maximalen Wissen, enthält. In dieser sind auch nicht in diesem Teilbaum benutzte CVs und/oder zusätzliche Properties für bereits vorhandene CVs enthalten. (Klassendiagramm s. Anhang auf Seite 81)

CVs haben durch die Properties die Möglichkeit, im *Mapping Dictionary* (s. auf Seite 20) und den NMTables ihre *Hometables* (URI Tabellen einer Klasse) zu bestimmen. Dazu wird eine Anfrage nach den möglichen Klassen an das *Mapping Dictionary* gestellt in welchen Tabellen die Klasse vorkommt. Dann wird überprüft ob es sich um eine *NM Tabelle* handelt, mit dem *NMTabs*, ist dies nicht der Fall, so ist die Tabelle die *HomeTable*.

CVs können auf verschiedene Weise benutzt werden. Als Variable, als Attribute und als Synonym. Wenn eine CV ein Synonym besitzt wird in der SQL Anfrage statt der üblichen Namenserzeugung, das Synonym eingesetzt. Dies wird verwendet, wenn eine neue Variable in *SPARQL* erzeugt wird, was *SQL* nicht kann.

5.2.1.1. PropertySet

Das PropertySet speichert die Beziehung zwischen der Property und den CVs. Jede CV speichert also eine Liste an Properties von Tripeln in denen diese CV als Subjekt vorkommt und welche CV als Objekt verwendet wurde. Außerdem wird jedem PropertySet noch zugeordnet, ob es *safe* oder *unsafe* ist. *Unsafe* sind alle PropertySets, die von dem rechten Kind eines Optional-Knotens nach oben gegeben werden. Bei *unsafe* kann das PropertySet nicht verwendet werden, um die Klasse der CV zu bestimmen. Das *unsafe* Flag muss bei der Property liegen, weil eine CV durchaus innerhalb wie außerhalb des *Optional* vorkommen kann. PropertySets werden beim Kopieren der Variable ebenfalls kopiert, da das *unsafe* Flag innerhalb des Optional-Kindes als *safe* betrachtet werden muss.

5.2.2. InformationSet

Jeder Knoten speichert ein InformationSet, das sowohl für die lokale Speicherung der benötigten Informationen als auch für die Kommunikation zwischen den Knoten verwendet wird.

Das InformationSet speichert die für den Knoten relevanten Teil der Anfrage. Die einzelnen Selects, Tabellen und Conditions werden in separaten Listen abgelegt. Dadurch ist das InformationSet auch in der Lage aus seinen enthaltenen Informationen eine SQL Anfrage zu erstellen. Dabei greifen sie rekursiv auf die InformationSets der Kinder ihres Knotens zu, um sie in als entsprechende Unterabfragen in die From-Klausel einzubauen. Um die gesamte Anfrage zu bekommen reicht es also, aus den Projektion-Knoten nach der Umwandlung seines InformationSets in eine SQL-Anfrage zu fragen. In Tabelle 5.1 sind die restlichen wesentlichen Methoden des InformationSets aufgelistet. Die vier add-Methoden fügen die jeweiligen Informationen dem InformationSet hinzu, wobei Duplikate vermieden werden. Hierbei wird nicht nur auf den Namen der jeweiligen der Tabelle, Selects, etc. geachtet, sondern auch auf die CV-Zugehörigkeit. Conditions, die zwar unterschiedlich auf-

gebaut sind, aber logisch äquivalent sind, werden dabei allerdings nicht zusammengefasst, da hierfür ein seperater Reasoner benötigt würde. Die Methode optimizeRemoveObsoleteTalbes entfernt Tabellen, die weder von Selects noch von Conditions benötigt werden. Diese entstehen größtenteils bei der Bildung von Union-Ausdrücken. Außerdem können Chain-Conditions, wie in Kapitel 5.2.5.3 beschrieben, mit der Methode optimizeCombineCondition zusammengefasst werden. Eine weitere komplexe Methode ist findUnions, die anhand der Selects oder Conditions in der Lage ist Tabellen, die einen Unions-Ausdruck bilden sollten, zu identifizieren. Diese werden in eine Union-Tabelle zusammengefasst und die nötigen Änderungen an Selects und Conditions durchgeführt.

Methode	Parameter	Rückgabe	Funktion
add	Set	-	Erweitert den Inhalt des eigenen InformationSets um den des übergebenen Sets
addSelect	Select	-	Fügt ein nicht bereits vorhandenes Select dem InformationSet hinzu
addTable	Table	-	Fügt eine Tabelle hinzu falls sie nicht bereits vorhanden oder Teil eines Union-Ausdruckes ist
addCond	Condition	-	Fügt eine Condition dem InformationSet hinzu, falls sie noch nicht vorhanden ist. (Dabei wird nicht auf Äquivalenz sondern nur exakte Gleichheit geprüft)
optimize Remove Obsolete Tables	-	-	Entfernt nicht mehr benutzte Tabellen aus dem InformationSet des Knotens
optimize Combine Condition	-	-	Ersetzt nicht notwendig Chain Conditions durch entsprechen bessere Lösungen
findUnions	kein	-	Findet und erstellt Unions wenn möglich

Tabelle 5.1.: Wichtige Methoden des InformationSets

5.2.2.1. Select

Das Select enthält primär die selektierten Spalten einer Tabelle, den Variablennamen mit dem dieser Wert selektiert wird und die Verknüpfung zum Tabellenobjekt der jeweiligen Tabelle.

Als sekundäre Attribute hat es noch fünf weitere Werte. Zum einen ein Flag, das angibt ob die Tabelle in der SQL-Anfrage mit aufgelistet werden soll oder die Spalte aus einer Unterabfrage kommt. Zum anderen Werte die bei Nulljoins (s. auf Seite 43) notwendig werden. Dazu gehört ein Flag, das speichert ob die Selektierung Nulljoins enthalten kann, damit später ein Case-Ausdruck für diese Spalte verwendet werden kann. Für diesen Ausdruck muss zusätzlich die alternative Spalte, der alternative Variablenname und die alternative Unterabfrage gespeichert werden.

5.2.2.2. Table

Die Table-Liste enthält alle für diesen und die höheren Abschnitte nötigen Table-Objekte. Ein Table-Objekt speichert die Tabellennamen und den Variablennamen mit dem die Tabelle umbenannt wird. Ebenfalls wird ein CreatedOf-Attribut angelegt und gespeichert welche CV dem Objekt zugeordnet ist und ob diese entfernbar ist oder nicht. Die Tabellennamen werden in einer Liste gespeichert, weil es mehrere Tabellen geben kann, wenn die verbundene CV mehrere Klassen beschreibt. Bei einem solchen Table-Objekt handelt es sich also um eine Union-Table (s. auf Seite 39).

Wenn nun ein Join oder eine andere Zusammenfassung von Informationen stattfindet, muss klar sein, welche Tabellen eine Union alles enthalten kann und je nach Situation entschieden werden, ob die Tabelle in der Union enthalten ist oder ob die Union auf diese Tabelle reduziert werden kann (Union kommt aus einem Optional). Das CreatedOf-Attribut wird benutzt, wenn in der From-Klausel anstelle einer Tabelle, eine Unterabfrage erstellt werden soll (dies ist meistens der Fall, außer bei BGPs). In solchen Fällen wird der Knoten dann gespeichert, dessen Anfrage als Tabelle benutzt werden soll. Dieser wiederum ruft dann seine toSQL-Methode auf und die komplette Unterabfrage wird dann anstelle des Tabellennamens eingefügt.

5.2.2.3. Conditions

Die Konditionen bestehen in einer Liste aus *Condition-Objekten*. In einer *SQL* Anfrage werden diese dann konjunktiv verbunden. Disjunktionen müssen innerhalb eines Condition-Objekts erstellt werden. Jeder einzelne Knoten selbst erstellt nur eine große *Condition* und über die Verbindung von unterschiedliche Knoten kommen keine disjunktiven Beziehungen in die Bedingungen. Die *Conditions* selbst können beliebig komplex und verschachtelt sein. Dafür gibt es 3 Unterarten von Conditions; *Simpel-*, *Complex1-* und *Complex2Condition*s (Klassendiagramm s. Anhang Abb. A.3).

Die *SimpleCondition* ist der grundlegende Baustein für Conditions und enthält deshalb nur eine CV oder ein Literal, während die Complex-Conditions jeweils aus einem Operator und einer oder zwei weiteren *Conditions* bestehen. So entsteht ein Baum aus *Conditions*, an dessen Blättern jeweils *SimpleConditions* stehen und in den höheren Knoten *ComplexCondition*s. Die *Complex1Conditions* sind einwertige Conditions, wie zum Beispiel der Negations-Operator, der vor eine beliebige Condition gestellt wird. Mit dem *Complex2Condition-Objekt* stehen alle weiteren gängigen Operatoren wie $\wedge$ oder $\vee$ zur Verknüpfung von zwei beliebigen Conditions zur Verfügung. Dieser Ansatz wurde von Jena[7] übernommen und ähnlich wie der Syntax Tree, nur den jeweiligen Bedürfnissen entsprechend rekonstruiert.

5.2.3. RequireSet

RequireSets sind Konstrukte, die alle benötigten Informationen eines Knotens zur Weitergabe an die Kinderknoten zusammenfassen. Folglich kommen sie nur in dem dritten Programmschritt zum Einsatz. Ein RequireSet speichert eine benötigte CV und einen Verweis auf den Knoten von dem der RequireSet ausgeht. Wenn mehrere CV benötigt werden, so werden die einzelnen RequireSets in einer Liste gesammelt und als solche weitergegeben. Hierfür dient die requestInformation-Methode, die jede Knote besitzt.

Die Kinderknoten bearbeiten die RequireSet hintereinander und versuchen die enthaltene CV in ihrer lokalen CVListe zu finden. Für RequireSets, auf die keine Antwort gefunden werden kann, werden neue RequireSets an die Kinder geschickt und ihre Antwort an den originalen Sender zurückgeschickt.

5.2.4. Nodes

5.2.4.1. Node

Node ist die Oberklasse aller Knoten und übernimmt die Aufgaben, die alle Knoten gemeinsam haben. Das heißt vor allem Methoden zur Verteilung und Anfrage von Informationen zu CVs, sowie allgemeine Umformungsmethoden wie die ScaleDown- und die toSQL-Funktion. Ein vollständiges Klassendiagramm für alle Node-Klassen kann im Anhang in Abb. A.2 gefunden werden. Die wichtigstens Methoden sind in Tabelle 5.2 zusammengefasst und werden nun weiter erläutert.

Die Methoden addCV und getCV sind simple Methoden, die die Verwaltung des InformationSets eines Knotens erleichtern. Die requestInformation-Methode wird von jeder Knoten-Unterklasse spezifisch implementiert und regelt die Informationsbeschaffung bei angeforderten CVs durch RequireSets.

Methode	Parameter	Rückgabe	Funktion
addCv	CV c	-	Fügt eine CV dem InformationSet des Knotens hinzu oder ergänzt eine vorhandene CV
getCV	String s	CV	Gibt die CV mit dem Namen s zurück, falls sie nicht vorhanden ist wird sie erstellt.
request-Information	RequiredSet r	Set	Leitet das RequiredSet an seine Kinder weiter. Gibt ein InformationSet mit der Selektierung zurück und fügt es sich selbst hinzu.
getInformation	-	-	Stellt die für diesen Knoten benötigten Informationen zusammen.
explore	String w, Expr e, selectedVars	String	Baut aus einer Expression und den selektierten Variablen eine gültige Bedingung
toSQL	Tiefe	String	Gibt die SQL Anfrage zu dem Knoten und seinen Unterknoten mit der angegeben Einrückung zurück
scaleDown	-	-	Fügt die Informationen mit dem darunter liegenden Knoten zusammen, so lange beide Knoten einwertig sind.

Tabelle 5.2.: Wichtige Methoden der Node-Oberklasse

Die toSQL Methode verwendet das jeweilige InformationSet des Knoten. Das Set enthält bereits alle Informationen in einer für SQL guten Sortierung. Die Methode verbindet nun mehrere Informationen miteinander und fügt die entsprechende Schlüsselwörter für die Anfragenstruktur ein. So kann das Select direkt in den Select-Block umgewandelt werden, das Table in den From-Block und das Condition in den Where-Block. Knoten wie Group

by, die besondere SQL Anweisungen haben, überschreiben oder erweitern diese Methoden entsprechend.

Die ScaleDown Methode findet nur bei Knoten Anwendung, die nur ein Kind haben. Hier werden dann die InformationSets so zusammengefasst, dass sie die Informationen von beiden Knoten enthalten. Dabei muss darauf geachtet werden, dass nach unten hin die Referenzen des Kindes, jedoch nach oben hin die Referenzen des Vaterknotens, genutzt werden. Dadurch können mehrere Knoten in der Abfrage nur noch als ein einzelner Select-From- Where-Block umgesetzt werden.

5.2.4.2. Tripel

Der Tripel-Knoten speichert sein Subjekt, Property und Objekt. Dabei überprüft es ob Subjekt bzw. Objekt eine Resource oder eine Literal ist.

Das Tripel hat im wesentlichen zwei Aufgaben: Zum einen erstellt es die CVs, die es selber enthält, zum anderen sucht es im Mapping Dictionary (s. auf Seite 20) nach den jeweils benötigten Tabellen und Selects für die SQL-Anfrage. Für diese erstellt es ein InformationSet und speichert sie dort ab. Die wichtigstens Methoden des Tripels sind in Tabelle 5.3 aufgelistet.

Die getBasicInformation-Methode wird in dem zweiten Programmschritt ausgeführt und sorgt für die Erstellung der CVs. Zudem wird dem Subjekt in der Liste knownProperties die Property mitgegeben und sofern eine inverse Property bekannt ist, diese dem Objekt. Diese CVs werden dann jeweils an den Vater-Knoten weitergeben, wo die Informationen der einzelnen Kinder zusammengefasst werden.

Der Kern eines Tripels stellt aber die findJoin-Methode dar. Diese wird ebenfalls noch im zweiten Programmschritt benutzt um alle benötigten Joins zwischen den SQL-Tabellen herzustellen, damit im dritten Programmschritt nur noch die entsprechenden Spalten dieser Tabellen selektiert werden müssen. Dafür werden die Hometables des Subjekts bestimmt und für alle Klassen des Subjekts überprüft in welche Tabelle der entsprechende Mapping Dictionary-Eintrag für diese Property zeigt. Ist die Tabelle keine NM-Tabelle und ist die Objekt-CV mehr als ein einfaches Attribut, so wird die Spalte aus dem MD-Eintrag mit den URI-Spalten aller Hometables der Objekt-CV gejoint und als Joincondition hinzugefügt. Gleiches gilt für Properties, die im MD-Eintrag als Inverses markiert sind, für diese werden jedoch entsprechend die Seiten getauscht.

Methode	Parameter	Rückgabe	Funktion
findJoin	-	Set	Erstellt die nötigen Joins über nicht-NM-Beziehungen
join	2 CV, Tabellen-name, Property	Set	Erstellt die Bedienungen um zwei CV über eine NM-Tabelle zu joinen
getBasic-Information	String c	Set	Liefert die Informationen für die CV c die dieses Tripel enthält zurück

Tabelle 5.3.: Wichtige Methoden der Tripel-Klasse

Ist die Tabelle des MD-Eintrags eine NM-Tabelle, so wird die join-Methode aufgerufen.

Diese findet über den Tabellennamen und die Property als LookupAttr der entsprechende Eintrag in der NMJ-Tabelle für das FkJoinAttr. Von der Subjekt-Seite aus, wird nun die URI-Spalte alle Hometables an die FkJoinAttr-Spalte der NM-Tabelle gejoint, während von der Objekt-Seite aus die URI-Spalten der Hometables an das LookupAttr der NM-Tabelle gejoint wird.

Des weiteren können auch SymmReified-Beziehungen in der findJoin-Methode identifiziert und behandelt werden, näheres hierzu in Kapitel 6.2.3.

5.2.4.3. Projection

Der Projection-Knoten hat die wohl simpelste, aber doch bedeutenste Aufgabe des gesamten Syntaxbaums. Da es aufjedenfall die Wurzel jedes Syntaxbaumes ist, sind die Informationen über CVs, die durch die getInformation-Methode erhalten werden, maximal. Daher wird die CVList des Projection-Knotens in den Baum propagiert und von den Kinder-Knoten als FullCVList gespeichert.

Außerdem werden die CVs, die für die endgültige Ausgabe benötigt werden, in der Liste selectedVariables gespeichert. Mit diesen fungiert der Projection-Knoten wie ein Motor für den restlichen Syntaxbaum. Im dritten Programmschritt beginnt der Projection-Knoten damit die selectedVariables in RequireSets zu verpacken und mit der requestInformation-Methode Anworten von seinem Kind anzufordern. Damit werden die Kinder angetrieben die benötigten Informationen zusammenzustellen und eigene Anforderungen jeweils für ihre Kinder zu erstellen.

5.2.4.4. BGP

Das BGP hat zwei wichtige Funktionen, zum einen besteht seine Aufgabe darin, zu erkennen welche CVs zusammengehören und die InformationSets zu verwalten. Das BGP erhält von jedem drinrt Tripel jeweils ein eigenes InformationSet und eigene CVs und führt diese zusammen. Da die InformationSets im Allgemeinen nicht disjunktiv sind, muss das BGP beim Zusammenfügen darauf achten, keine Duplikate zu erstellen (dies ist eine Grundeigenschaft der Node-Klasse). Die Erstellung dieser Informationen und Variablen liegt allerdings bei den Tripeln.

Zum anderen muss es auf Requests der höheren Knoten reagieren. Im Gegensatz zu anderen Node-Klassen im Allgemeinen kann das BGP diese Anfragen jedoch nicht einfach weiter an seine Kinder leiten, da diese nur Tripel sind. Da das BGP aber die erste Instanz ist, die Informationen über CVs zusammenfasst, ist es auch in der Lage zu entscheiden, ob es ein RequestSet beantworten kann. Wenn ja, bestimmt es die für diese Anfrage nötigen Selects und Tabellen und sendet sie als ein InformationSet wieder rekursiv zum Anfragenden zurück.

In Abb. 5.1 wird die Auswertung eines BGPs mit ein paar simplen Tripeln gezeigt, die das Verhalten bei der Erstellung der Selects verdeutlichen soll. Die Tabellen behalten eine Referenz zu den CVs, die zu ihrer Erstellung geführt haben, damit Attribute von der selben CV auch die selbe Tabelle benutzen. In dem Beispiel wird die benötigte Tabelle City in c_City umbenannt, da sie von der CV *c* abhängt. Die erstellten Selects speichern ebenfalls, zu welcher CV sie gehören, damit sie zum einen eindeutige Namen besitzen und zum anderen damit RequireSet-Anfragen mit den korrekten Select-Ausdrücken beantwortet wer-

den. Das zweite Tripel „?c :name ?cn" führt dazu, dass die CV *cn* auf die Spalte „name" von der/die Tabelle/n der CV *c* referenziert. Da cn von Projection1 mit einem RequireSet angefordert wird, wird die Spalte folglich als c_City.name selektiert, aber zu City_cn für die eindeutige Benennung umbenannt. Der Variablenname City_cn wird an Projection1 geleitet, damit dieser diese für die CV *cn* ebenfalls selektieren kann. Da die Antwort aus dem Kind BGP2 kam, wird folglich BGP2.City_cn selektiert. Dieses Verhalten ist für das dritte Tripel äquivalent.

Für die Auswertung der Anfrage wurde auf die scaleDown-Methode verzichtet, damit der Fluss der Informationen zwischen den einzelnen Knoten besser in der ausgegeben Anfrage dargestellt wird. Die Auswirkungen der scaleDown-Methode sind in der Abb. 5.2 und den zugehörigen Erläuterungen, zu finden.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?cn ?cp
WHERE { ?c a :City .
        ?c :name ?cn.
        ?c :population ?cp . }
-------Syntaxtree------
(Projection1( cn cp )

    (BGP2
        (Triple3( c type City )
        (Triple4( c name cn )
        (Triple5( c population cp )
    )

)
----------SQL Anfrage---------
Select BGP2.City_cn as Result_cn, BGP2.City_cp as Result_cp
From (

    Select c_City.name as City_cn, c_City.population as City_cp
    From City c_City
    ) BGP2
```

Abbildung 5.1.: test-bgp.sparql ohne ScaleDown

5.2.4.5. Filter

In Abbildung 5.2 ist eine einfache Anfrage mit Filter gegeben. Gesucht werden alle Städte mit ihrer Einwohnerzahl, sofern diese über 1 Millionen liegt. Der Filter-Knoten speichert das Expr-Objekt aus dem Jena-Syntaxbaum, indem der Ausdurck „p > 1000000" enthalten ist. Mit der explore-Funktion werden alle CVs, die in dem Expr-Objekt enthalten sind, gesammelt und RequireSets für diese an den Kindknoten geschickt. In diesem Fall wird nur die CV *p* von dem BGP3 gefordert, wodurch Filter2 den Variablennamen von p erhält.

Danach wird das Expr-Objekt in ein Condition-Objekt umgewandelt. Dabei entsteht eine Complex2Condition mit dem Operator „>" und zwei SimpleConditions als Kinder. Das linke Kind enthält nur die CV *p*, während das rechte Kind das Literal 1000000 enthält. Für die CV *p* wird gleichzeitig der passende Variablenname „BGP3.population" eingesetzt, da sie vom BGP3 erhalten wurde.

Wie man an der Tabellenbezeichnung „BGP3" sieht, wird hier der Filter noch als eigenständiger Knoten behandelt, der eine eigene Anfrage mit dem Kind als Unterabfrage namens BGP3 erstellt. Die Abbildung 5.2 zeigt aber, dass keine Unterabfrage entsteht und die Informationen des Filter-Knotens dem BGP-Knoten hinzugefügt werden. Diese Umwandlung wird durch die Methode scaleDown durchgeführt, damit viele Knoten nicht schnell zu einer unübersichtlichen Anzahl an Unterabfragen übersetzt werden. Dabei muss darauf geachtet werden, dass die Tabellenbezeichnungen von selektierten Spalten, sowie der Conditions, des Filter-Knotens an die Bezeichnungen des BGPs angepasst werden.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?n ?p
WHERE {

    ?x a :City.
    ?x :population ?p.
    ?x :name ?n.
    FILTER (?p > 1000000) . }

-------Syntaxtree------
Projection1( n p ) (

    Filter2 (
        BGP3 (
            Triple4( x type City )
            Triple5( x population p )
            Triple6( x name n )

)))
-------SQL Query-------
Select x_City.name as Result_n, x_City.population as Result_p
From City x_City
Where (x_City.population > 1000000)
```

Abbildung 5.2.: Filter.sparql

5.2.4.6. Join

Der Join-Knoten ist ein Hilfskonstrukt, das nur benötigt wird, wenn die SPARQL-Anfrage aufgespalten konstruiert ist. Nehmen wir zum Beispiel die Anfrage aus Abb. 4.2, so entsteht in dem zugehörigen Syntaxbaum kein Join-Knoten. Verschieben wir den Optional-Block wie in Abb. 5.3 dargestellt, so entsteht ein Join-Knoten, obwohl die Anfrage äquiva-

lent ist.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?xn ?cn ?ln
WHERE {
    ?x :name ?xn.
    ?x :capital ?c.
    Optional{
        ?c :locatedAt ?l .
        ?l :name ?ln
    }
    ?c :name ?cn .
    ?c :population ?p.
    Filter (?p > 1000000)
}
-------Syntaxtree------
Projection1( x xn cn ln ) (
    Filter2 (
        Join3 (
            LeftJoin4 (
              BGP5 (
                Triple6( x name xn )
                Triple7( x capital c )
              )
              BGP8 (
                Triple9( c locatedAt l )
                Triple10( l name ln )
              )
            )
            BGP11 (
              Triple12( c name cn )
              Triple13( c population p )
            )
        )
    )
)
```

Abbildung 5.3.: Alternative Form des einführenden Beispiels

Der Join-Knoten sorgt also nur dafür, dass die Ergebnismengen seiner beiden Kinder zusammenführt wird. Dies wird durch den Join der CVs, die in beiden Kindern vorkommen, ermöglicht. Dieses Verhalten kann aber nicht nur in dem Join-Knoten vorkommen, sondern wird für alle Knoten benötigt, die mehr als ein Kind besitzen. Daher wird hier auf die Methode namens getJoin eingegangen, die zwar für alle solche Knoten etwas anders implementiert ist, aber die gleiche grundlegende Vorgehensweise besitzt.

Der getJoin-Methode werden zwei CVs übergeben, die in beiden Kindern vorkommen und den selben Namen tragen. Dadurch ist gegeben, dass sie die gleich SPARQL-Variable beschreiben, aber nicht unbedingt, dass sie über das selbe Wissen über diese verfügen. Für beide Variablen wird nun ein RequireSet vom Vater-Knoten an die Kinder gestellt. Die Antworten werden als ein equi-Join der Form $Leftchild.var - column = Rightchild.var - column$ verbunden und der Joincondition hinzugefügt.

5.2.4.7. Left Outer Join

Der Leftjoin-Knoten arbeitet grundsätzlich wie der Join-Knoten, doch können hier ein paar Sonderfälle auftreten. Entscheidender Unterschied ist jedoch, dass die beiden Unterabfragen, die von den Kindern erhalten werden, in der toSQL-Methode nicht mit einem normalen Join (','), sondern mit einem Left Outer Join verbunden werden. Als Nebeneffekt wird der Condition-Abschnitt des InformationSets zudem als On-Abschnitt anstatt eines Where-Abschnitts in der SQL-Anfrage umgesetzt.

Grundsätzlich werden die beiden Mengen aus den Kindern, wie in einem Join-Knoten, mit der getJoin-Methode zusammengeführt. In Abb. 5.4 wird eine Anfrage mit zwei Optional-Blöcken gezeigt, an der die verschiedenen Fälle bei der Auswertung erläutert werden. Die Anfrage sucht alle Paare von Städten, für die jedoch der Fluss, sofern sie an einem liegen, ausgegeben werden soll. LeftJoin3 ist ein normaler Left Outer Join, der mit Hilfe der getJoin-Methode ausreichend behandelt wird. Die CV x kommt sowohl in BGP4 als auch BGP9 vor und wird daher mit $BGP4.City\_x\_URI = BGP9.City\_x\_URI$ gejoint.

Der LeftJoin2 mit den Kindern LeftJoin3 und BGP12 stellt aber ein Sonderfall dar. Wie in Kapitel 4.3.6 angesprochen ist es nun möglich, dass der Wert, der an eine CV W gebunden wird, in beiden Kindern nicht sicher ist. Diese Eigenschaft wird mit der findCases-Methode überprüft und entsprechend die Joincondition erweitert. Diese Methode ist auch für den Join-Knoten verfügbar, da dort auch die selbe Situation auftreten kann. Die getJoin-Methode hat bereits für alle CVs, die in beiden Kindern vorkommen, ein equi-Join erstellt. In diesem Fall ein Join der URI-Spalten der CV y wie im LeftJoin3, aber auch der Join für die CV W: $(LeftJoin3.River\_W\_URI = BGP12.River\_W\_URI)$. Nun werden diese CVs darauf überprüft, ob sie durch eine als unsafe markierte Property selektiert wurden. Ist dies für beide der Fall, wie für die CV w, wird die Joincondition mit folgenden Ausdruck erweitert:

$or\ (LeftJoin3.River\_W\_URI\ is\ null\ and\ BGP12.River\_W\_URI\ is\ not\ null)$
$or\ (LeftJoin3.River\_W\_URI\ is\ not\ null\ and\ BGP12.River\_W\_URI\ is\ null)$
$or\ (LeftJoin3.River\_W\_URI\ is\ null\ and\ BGP12.River\_W\_URI\ is\ null)$

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?xn ?yn ?W
WHERE {

    ?x a :City.
    ?x :name ?xn.
    ?y a :City.
    ?x :name ?yn .
    OPTIONAL { ?x :locatedAt ?W.
               ?W a :River}.
    OPTIONAL { ?y :locatedAt ?W.
               ?W a :River}

}
-------Syntaxtree------
Projection1( xn yn W ) (

    LeftJoin2 (
        LeftJoin3 (
            BGP4 (
              Triple5( x type City )
              Triple6( x name xn )
              Triple7( y type City )
              Triple8( x name yn )
            )
            BGP9 (
              Triple10( x locatedAt W )
              Triple11( W type River )
            )
        )
        BGP12 (
            Triple13( y locatedAt W )
            Triple14( W type River )

)))
```

Abbildung 5.4.: Verschiedene Arten von Left Outer Join in einer Anfrage

Zur Erinnerung: Nun können vier Fälle beim Binden von Werten an diese CV vorkommen:

1. Sowohl LeftJoin3 als auch BGP12 binden ein Ergebnis an die CV W $\Rightarrow$ Die Werte müssen gleich sein.
2. Nur LeftJoin3 bindet ein Ergebnis $\Rightarrow$ Der Wert aus LeftJoin3 muss genommen werden.
3. Nur BGP12 bindet ein Ergebnis $\Rightarrow$ Der Wert aus BGP12 muss genommen werden.
4. Keiner binder ein Ergebnis $\Rightarrow$Selektiere den Null-Wert.

Daher muss der LeftJoin2 beide Spalten in seinem Select verbinden. Dies geschieht mit einem Case-Ausdruck, der wie folgt gebildet wird:
CASE WHEN LeftJoin3.River_W_URI ist null THEN BGP12.River_W_URI
ELSE LeftJoin3.River_W_URI END as River_W_URI
Um diesen Ausdruck erstellen zu können, wird in der findCases-Methode ein spezielles Select-Objekt mit einer alternativen Spalte erstellt, das mit der toSQL-Methode zu dem Ausdruck oben umformuliert wird.

5.2.4.8. Group By

Der Group By-Knoten speichert zusätzlich zu den Node-Informationen das OpGroup-Objekt, das aus dem Jena-Syntaxbaum enthalten wurde, und eine Liste an CVs namens createdFilterCV, die innerhalb des OpGroups durch Umbenennung neu entstanden sind. Durch den Group By-Block in einer Anfrage, können ab diesem Knoten nur noch CV selektiert werden über die gruppiert wird und in Kombination mit einer Aggregationsfunktion. Da Aggregationsfunktionen in dem Select-Block in der benutzten Version von Jena noch nicht funktionieren, muss eine Liste namens createdCV angelegt werden, die festhält, welche CVs bei einem requestInformation noch beantwortbar sind.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?sum
WHERE {

    ?x a :City.
    ?x :longitude ?lo .
    ?x :latitude ?la .

}
Group by (?lo + ?la as ?sum)
----------SQL Anfrage---------
Select (City_lo + City_la) as sum
From (

    Select x_City.longitude as City_lo, x_City.latitude as City_la
    From City x_City

)
GROUP BY (City_lo + City_la)
```

Abbildung 5.5.: Group By mit Umbenennung

Bei der Konvertierung eines Group By-Knotens in eine SQL-Anfrage stellt sich jediglich die Problematik von umbenannten Ausdrücken in eine CV. In Abb. 5.5 wird ein solches Group By gezeigt, bei dem der Ausdruck ?*lo*+?*la* in der CV *sum* zusammengefasst wird. Die CV *sum* steht folglich sowohl in der createdCv- als auch in der createdFilterCV-Liste. Bei der Erstellung wird ihr über die setSynonymfor-Methode der entsprechende Ausdruck mitgegeben. Dadurch wird ?*sum* eine Synonym-Variable und wird in allen folgen-

den Schritten, wenn sie benötigt wird mit ihrem Synonym ersetzt. In der Abb. 5.5 passiert dies in dem Select-Block der SQL-Anfrage.

5.2.5. Vereinfachung der Anfragen

Die rekursive Konstruktion der Anfrage führt zu einem korrekten Ergebnis, doch ist sie meistens noch komplexer als notwendig. Am deutlichsten sieht man dies daran, dass eine Anfrage sehr stark verschachtelt ist ohne Optimierung, da jeder Knoten des Syntaxbaumes sein eigenen *SELECT, FROM, WHERE* Block erstellt. Dies sollte keine großen Auswirkungen auf die Laufzeit haben, macht die Anfragen aber sehr unübersichtlich. Des Weiteren werden auch noch Tabellen entfernt, die nicht mehr benötigt werden und Chain Conditions optimiert.

5.2.5.1. Entfernung von übersprungenen Tabellen

Beim Erstellen von Union-Tables werden die ursprünglichen Tabellen häufig unnötig, wenn sie weder in einem SELECT- noch einem WHERE- Block benutzt werden. Letztendlich schränken sie die SQL Anfrage in ihrer Gültigkeit nicht ein, machen sie aber unübersichtlich und unter Umständen langsamer. Deshalb wird beim Optimierungsschritt „optimizeRemoveObsoleteTables" aufgerufen. Diese Funktion überprüft bei den Informationssets aller Knoten die Tabellen auf ihre Relevanz. Tabellen die keine Referenzen mehr haben werden entfernt, ähnlich wie bei einem Garbage - Collector bei Programmiersprachen.

5.2.5.2. Entfernung unnötiger Verschachtelungen

Da jeder Knoten in seinem *toSQL* Schritt eine komplettes Set aus SELECT, WHERE und FROM erzeugt und die tiefer liegenden Knoten jeweils im FROM unterbringt, werden die endgültigen Anfragen in der Regel unnötig tief geschachtelt und unübersichtlich. Bei Knoten mit mehr als einem Kind (BGP's, Left Outer Joins und Joins) sind diese Verschachtelungen allerdings notwendig. Bei allen anderen Fällen ist es jedoch sinnvoll die InformationSets der Knoten miteinander zu verschmelzen, um ein paar Schichten der Anfrage zu entfernen.

Dazu werden die Variablennamen von höher liegenden Knoten eine Tiefe runter gezogen und ersetzen dann die Namen der tiefer liegenden Schritte. So wird beispielsweise aus $Node.x\_URI\ as\ x\_URI$ ein $Node.x\_URI\ as\ Result\_x$, wenn die Projektion einen Knoten heruntergezogen werden kann. Bei der sogenannten *scaleDown-Funktion* muss neben dem SELECT auch die WHERE Klausel angepasst werden. Die Variablennamen der Bedingungen müssen geändert werden und sie werden in den Knoten tiefer verlagert. Dies wird für jeden Knoten ausgeführt, der nur ein einzelnes Kind hat, außer bei BPGs die nur ein Triple beinhalten, hier macht dies natürlich keinen Sinn, da sie in der toSQL Funktion die letzten erfragten Glieder sind, sowie bei Group by-Knoten (s. auf Seite 37).

5.2.5.3. Optimierung von Chain Condition

Chain Conditions sind Conditions, die zwei Tabellen über eine dritte Verbindungstabelle verknüpfen. In einigen Fällen ist es nicht notwendig für eine Chain Condition, dass die

verbindende Tabelle benutzt wird. So ist es beispielsweise für die Anfrage Abbildung 5.6, hier kann die URI von *?cou :capital* und die von *?o :hasHeadq* direkt miteinander verglichen werden, ohne die City Table zu verwenden.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?cou ?o
WHERE  {
    ?cou a :Country .
    ?c a :City .
    ?o a :Organization .
    ?m a :Membership .
    ?cou :capital ?c .
    ?o :hasHeadq ?c .
    ?m :ofMember ?cou.
    ?m :inOrganization ?o
}
```

Abbildung 5.6.: Chain Condition SPARQL

Damit dies in einer SQL Anfrage umgesetzt werden kann, muss zunächst einmal überprüft werden, ob die CV von dem zu überspringenden Elemente noch anderweitig benötig wird, z.B. wenn man noch den Namen der Hauptstadt haben möchte. Ist dies nicht der Fall, so können die betreffenden Conditions im InformationSet gesucht und entfernt werden sowie eine neue Condition hinzugefügt werden, in der die beiden zu verbindenden Elemente direkt verbunden sind.

```
Select cou_Country.URI as Result_cou, o_Organization.URI as Result_o
From Country cou_Country,
     Organization o_Organization,
     Membership m_Membership,
     City c_City
Where
    (m_Membership.ofMember = cou_Country.URI) and
    (m_Membership.inOrganization = o_Organization.URI) and
    (cou_Country.capital = c_City.URI) and
    (o_Organization.hasHeadq = c_City.URI)
```

Abbildung 5.7.: Unoptimierte Chain Condition

Die SQL Anfrage aus Abbildung 5.7 sollte dann wie in Abbildung 5.8 aussehen.

```
Select cou_Country.URI as Result_cou, o_Organization.URI as Result_o
From Country cou_Country,
     Organization o_Organization,
     Membership m_Membership
Where

    (m_Membership.ofMember = cou_Country.URI) and
    (m_Membership.inOrganization = o_Organization.URI) and
    (cou_Country.capital = o_Organization.hasHeadq)
```

Abbildung 5.8.: Chain Condition SQL

6. Anwendung

6.1. Verwendung

Für die Verwendung des Query Converters wird grundsätzlich nur eine durch RDF2SQL erstellte Datenbank benötigt. Zudem müssen in der jdbc.props Datei die korrekten Login-Informationen für diese Datenbank eingetragen sein. Die Main-Klasse heißt Converter.java und kann mit folgenden Parametern aufgerufen werden:

Parameter	Variable	Funktion
-d	*Datenbank-Url*	Angabe des Pfads zur Verbindung mit der Datenbank
-q	*Dateiname*	Angabe der Datei, die die Query enthält
-o	*Dateiname*	Angabe der Datei für die Output-Speicherung
-t	-	Zeigt die Struktur der internen Syntaxbäume
-f	*[-1;0;1;2]*	Bestimmt die Formatierung des Outputs
-op	-	Zeigt die Optimierungsschritte
-si	*[0;1]*	Bestimmt ob die jedigliche Ausgabe sofort oder erst nach vollständiger Beendigung des Programms geschieht

Tabelle 6.1.: Konsolenparameter

Das Ergebnis der Anfrage wird ohne Angabe des „-o“-Parameters direkt in der Konsole angezeigt, ansonsten in einer Textdatei gespeichert. Für das Ausgeben der Ergebnisse der erstellten Anfrage können verschiedene Stufen der Formatierung gewählt werden:

- Stufe -1: die erstellte Anfrage wird nicht an die Datenbank gestellt, sondern lediglich die Anfrage ausgegeben
- Stufe 0: die Ergebnisse der Anfrage werden nur durch Tabs getrennt
- Stufe 1: die Datensatz-Ausgaben für die Spalte werden jeweils auf die maximale Spaltenlänge mit Leerzeichen ergänzt
- Stufe 2: erstellt eine Tabelle, die aus graphischer Sicht optimiert formatiert ist.

Die Formatierung für Stufe 2 braucht jedoch eine Weile, weil die Ergebnisse nicht sofort angezeigt werden können. Es muss einmal über alle Datensätze iteriert und diese im Hauptspeicher gehalten werden. Daher wird empfohlen diese Stufe nicht für Ergebnisse mit über 1000 Datensätze zu verwenden.

6.2. Beispiele

In diesem Kapitel wird anhand verschiedener Beispiele der Programmablauf genauer vorgestellt. Das erste Beispiel behandelt einen allgemeinen Fall mit dem die Interaktion der einzelnen Knoten deutlich werden soll. Die darauf folgenden Beispiele legen den Fokus jeweils auf eine spezielle Besonderheit der Konvertierung, die die Entwicklung des Programms erschwert haben. Alle verwendeten Anfragen sind im Programmpaket verfügbar.

6.2.1. Ausführliches Beispiel

In diesem Beispiel wird die Standard-Anfrage aus Abb.6.1 behandelt, die für jede Stadt ihren Namen, die Einwohnerzahl und sofern vorhanden das Gewässer an der sie liegt ausgibt. Damit die Ergebnismenge nicht zu groß wird, werden nur Städte gesucht deren Einwohnerzahl über 1 Million beträgt. Man sollte beachten, dass eine Stadt mehrmals ausgegeben werden kann, sofern sie an verschiedenen Gewässern liegt.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?xn ?xp ?l
WHERE {

    ?x a :City.
    ?x :name ?xn.
    ?x :population ?xp.
    Optional{ ?x :locatedAt ?l }.
    Filter(?xp > 1000000)
    }
```

Abbildung 6.1.: ausfuehrlichesbeispiel.sparql

Für diese Anfrage erhalten wir von Jena den Syntaxbaum aus Abb.6.2.

Dieser Syntaxbaum wird an den Treebuilder übergeben der mithilfe des TreeWalker und des TreeVisitor die Nodestruktur erstellt. Wir erhalten also einen Projection-Wurzelknoten, der als Kind einen Filter hat, der als Kind einen Left Outer Join hat, der als Kinder zwei BGPs hat, die wieder drei bzw. ein Tripel als Kinder haben. Das Left Outer Join ist in der SPARQL Syntax das Optional und die Tripel gleicher Stufe bilden zusammen immer erst ein BGP bevor ein anderer Knoten kommt. Deshalb entstehen bei dieser Anfrage auch zwei BGPs, einmal für außerhalb und einmal für innerhalb des Optionals.

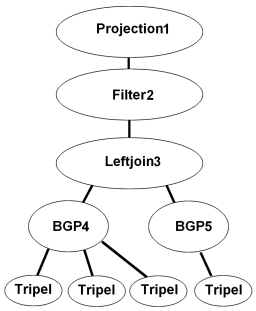

Abbildung 6.2.: SyntaxTree der Anfrage

Nun werden die CVs erstellt, rekursiv werden die Informationen von den Kindern an die Eltern weitergegeben und wir erhalten in der Wurzel folgende Informationen über die CVS:

CV	Klasse	Known Properties	is Safe	Type
x	City	Name, Population, LocatedAt	$\surd$	Resource
xn	Attr. von x	keine	$\surd$	Literal
xp	Attr. von x	keine	$\surd$	Literal
l	Sea, Lake, River	locatedAt_Inv	$\times$	Resource

Tabelle 6.2.: Eigenschaften der CVs

Nach der Erstellung der CVs beginnt das Erstellen der Informationssets. Dazu wird in den Blättern des Baumes, genau genommen allerdings den BGPs, da Tripel nicht direkt an der Informationssuche beteiligt sind, angefangen und sich dann hoch gearbeitet.

- **BGP 4**

  Das erste Tripel "?x a :City." liefert uns keine weiteren Erkenntnisse, die nicht schon in der CV Erstellung benutzt worden wären und wird deshalb nicht weiter bearbeitet. Jedoch wird für die Tripel "x :name xn" und "x :population xp" für alle Klassen von x, in diesem Fall nur City, geschaut, in welchen Tabellen und Spalten die Properties zu finden sind und ob ein Join mit xn oder xp entsteht. Da xn oder xp nur in Tabellen vorkommen, die zu den Hometables von x gehören (City), entsteht kein Join. Auch Union-Möglichkeiten gibt es hier keine.

- **BGP8**

  Überprüft für das Tripel "x locatedAt l" für alle Klassen von x in welchen Tabellen+Klassen die Property "locatedAt" zu finden ist. Im MD erhält man dann folgendes Ergebnis:

Tablename	lookupattr	inv
locatedAt	Water	false

  Daraus wird geschlossen, dass locatedAt nicht in die Hometables von x gehört, sondern eine eigene Tabelle ist und ein Join nötig ist.

  Für den Join wird eine NM Tabelle hinzugezogen. Für alle Klassen von x wird nach dem FkJoinAttr im NMJ für Tablename = "locatedAt" und lookupattr = "Water" nachgeschaut. Gefunden wird City, also ist die Spalte mit der gejoint werden muss die Spalte City. Aus dieser Information kann dann geschlossen werden, dass die Table "locatedAt" und die Joincondion "x_City.URI = x_locatedAt.City" ins Informationsset aufgenommen werden muss. Nun wird das Objekt des Tripels beachtet und es wird überprüft. Es ist von der Klasse Water und hat damit drei Unterklassen, Sea, Lake und River. Die entsprechenden Joinconditions werden dann mit Hilfe des MDs erstellt: „l_Sea.URI = x_locatedAt.Water", „l_Lake.URI = x_locatedAt.Water", „l_River.URI = x_locatedAt.Water". Die benötigten Tabellen und Konditionen werden nun dem InformationSet hinzugefügt. Anschließend wird nach Union gesucht und Sea/Lake, Lake/River und River/Sea gefunden. Daraus kann eine Union gebildet (Abb.6.3) werden und diese wird ebenfalls dem InformationSet hinzugefügt. Die nun überflüssigen Tabellen werden allerdings erst zu einem späteren Zeitpunkt entfernt, auch wenn sie nicht mehr benutzt werden.

```
Sea l_Sea, Lake l_Lake, River l_River
          ⇓
      (Select URI From Sea
 UNION Select URI From Lake
 UNION Select URI From River) l_Union
```

Abbildung 6.3.: Umwandlung in eine Union für Waterklassen

Nach der Erstellung einer Union werden nun alle Vorkommnisse der entsprechenden Variable auf die Union umgeschrieben.

- **Leftjoin3**

  Das Left Outer Join vergleicht nun seine Kinder BGP4 und BGP8. Für jede CV, die in beiden vorkommt, wird nun die URI-Spalte miteinander gejoint. Also in diesem Fall ist es x, und es wird die Kondition "BGP4.City_x_URI = BGP8.City_x_URI" in das Informationsset des Left Outer Joins aufgenommen.

- **Filter2**

  Filter identifizieren alle CV die in ihrer Bedingungen vorkommen und fordern die Selektierung von ihnen bei ihrem Kind an. In diesem Fall wird also von LeftJoin3 die CV xp verlangt. LeftJoin3 hat aber keine solche Variable bereits selektiert, also wird die Anfrage an seine Kinder BGP4 und BGP8 weitergeleitet. BGPs können die Anfragen nicht mehr weiterleiten und falls sie eine solche Variable nicht kennen, geben sie zurück, dass sie diese nicht kennen, sowie es BGP8 tun muss da es nur x und l kennt. BGP4 jedoch hat eine CV xp. Im MD wird überprüft um welche Spalten es sich handelt, in diesem Fall um City.population, und fügt diese Selektierung seinem Informationsset hinzu und leitet die Information an seinen Elternknoten weiter. LeftJoin3 erkennt, von welchem Kind Informationen kommen und von welchem nicht und selektiert BGP4.City_population in seinem InformationsSet und leitet es an den Filter zurück. Der Filter kann nun selektieren und die Bedienungen "LeftJoin3.City_population = 1.000.000" hinzufügen.

- **Projection1**

  Projection1 fordert die Variablen xn, xp und l von Filter2. Filter2 findet xp und gibt City_xp zurück. Die anderen beiden CV werden jedoch nicht in dem Filter benötigt und die Forderung muss weitergereicht werden an die tiefer liegenden Knoten. Leftjoin3 kennt sie ebenfalls nicht und gibt weiter zu seinen Kindern. BGP4 findet xn und gibt x_City.Name als City_xn nach oben. BGP8 findet l und gibt locatedAt.Water als locatedAt_l nach oben.

Durch den Optimizier werden dann noch die Tabellen, die nicht selektiert oder als Bedingungen gebraucht werden, entfernt. Also Lake, River und Sea von BGP8. Jetzt sind alle Informationen, die benötigt werden um die Anfrage zu erstellen, vorhanden. Jeder Knoten wandelt nun seine Informationen in eine einzelne SQL-Anfrage um und fügt seine Kinder als Unterabfragen ein. Das BGP erstellt natürlich keine Unterabfragen, sondern stellt die konkreten Tabellen.

——-SQL Query——-

```
Select Filter2.City_x_URI as Result_x,
       Filter2.City_xn as Result_xn,
       Filter2.City_xp as Result_xp,
       Filter2.l_URI as Result_l
       Filter2.ln_name as Result_ln
From (

    Select LeftJoin3.City_xp as City_xp,
           LeftJoin3.City_x_URI as City_x_URI,
           LeftJoin3.City_xn as City_xn,
           LeftJoin3.l_URI as l_URI,
           LeftJoin3.ln_name as ln_name
    From (
        Select BGP4.City_xp as City_xp,
               BGP4.City_x_URI as City_x_URI,
               BGP4.City_xn as City_xn,
               BGP8.l_URI as l_URI,
               BGP8.ln_name as ln_name
        From (
            Select x_City.URI as City_x_URI,
                   x_City.population as City_xp,
                   x_City.name as City_xn
            From City x_City
        ) BGP4
        Left Outer Join (
            Select x_City.URI as City_x_URI,
                   l_Union.URI as l_URI,
                   l_Union.name as ln_name
            From locatedAt x_locatedAt, City x_City,

              ( (Select URI URI, name name from Lake) Union
                (Select URI URI, name name from Sea)  Union
                (Select URI URI, name name from River) ) l_Union
            Where (x_City.URI = x_locatedAt.City)
              and (l_Union.URI = x_locatedAt.Water)
        ) BGP8
        On (BGP4.City_x_URI = BGP8.City_x_URI)
    ) LeftJoin3
    Where (City_xp > 1000000)

) Filter2
```

Abbildung 6.4.: Anfrage bevor sie von scaleDown bearbeitet wurde

Wie in Abb. 6.4 gezeigt wird, ist die erstellte Anfrage sehr groß und verwendet sehr häufig Unterabfragen, obwohl diese nicht unbedingt benötigt werden. Daher werden als letztes durch die scaleDown-Methode alle Anfragen von Knoten mit nur einem Kind zusammengeführt. So werden Projection1 und Filter2 zusammengefasst und danach Filter2

mit Leftjoin3. Die Informationsset werden dabei zusammengeführt und die Variablenname angepasst. Wir erhalten letztlich als Ergebnis die Anfrage wie in Abb. 6.5 gezeigt. Sie ist wesentlich übersichtlicher.

——-SQL Query——-

```
Select BGP4.City_xn as Result_xn,
       BGP4.City_xp as Result_xp,
       BGP8.l_URI as Result_l,
       BGP8.ln_name as Result_ln
From  (

    Select x_City.URI as City_x_URI,
           x_City.population as City_xp,
           x_City.name as City_xn
    From City x_City
    ) BGP4
    Left Outer Join (
        Select x_City.URI as City_x_URI,
               l_Union.URI as l_URI,
               l_Union.name as ln_name
        From locatedAt x_locatedAt, City x_City,
           ( (Select URI URI, name name from Sea)  Union
             (Select URI URI, name name from Lake) Union
             (Select URI URI, name name from River) ) l_Union
        Where (x_City.URI = x_locatedAt.City)
          and (l_Union.URI = x_locatedAt.Water)
    ) BGP8

On (BGP4.City_x_URI = BGP8.City_x_URI)
Where (BGP4.City_xp > 1000000)
```

Abbildung 6.5.: Anfrage nach dem Optimierungsschritt

6.2.2. Komplexere Group by Anfrage

In diesem Beispiel wird eine Anfrage behandelt, die alle Besonderheiten eines Group By-Knotens beinhaltet. Wie in Abbildung 6.6 zu sehen ist wird eine Funktion mit zwei SPARQL-Vars als eine neue Variable umbenannt und über diese gruppiert. Zudem wird auf der gruppierten CV eine Aggregationsfunktion ausgeführt und mit diesem Ausdruck eine Bedingung gestellt.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?sum
WHERE  {

    ?x a :City.
    ?x :longitude ?lo .
    ?x :latitude ?la .

}
Group by (?lo + ?la as ?sum)
Having (count(?sum) > 5 )
-------Syntaxtree------
Projection1( sum ) (

    Filter2 (
        Group3 (
            BGP4 (
              Triple5( x type City )
              Triple6( x longitude lo )
              Triple7( x latitude la )
            )
        )
    )

)
```

Abbildung 6.6.: GroupBy.sparql

Die Anfrage selber sucht nach Gruppen von Städten, die mit ihren Längen- und Breitengraden die gleiche Summe ergeben. Diese Gruppen müssen mindestens 6 Städte beinhalten um ausgegeben zu werden. Diese Anfrage hat natürlich keinen realen Zweck und dient nur zur Demonstration des Ablaufs. Wie in der Abbildung des Syntaxbaumes zu sehen ist, wird die Having-Klausel getrennt vom Group by-Knoten behandelt indem diese als einfacher Filter-Knoten, der dem Group by übergeordnet ist, umgewandelt wird.

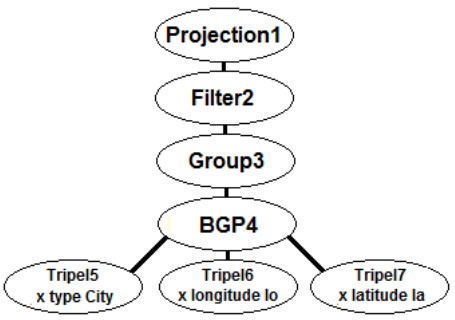

Abbildung 6.7.: Syntaxbaum von GroupBy.sparql

Durch den zweiten Schritt lassen sich dann folgende Informationen über die CVs gewinnen:

CV	Klasse	Known Properties	is Safe	Type	Synonymfor
x	City	longitude, latitude	$\surd$	Resource	$\times$
lo	Attribut von x	keine	$\surd$	Literal	$\times$
la	Attribut von x	keine	$\surd$	Literal	$\times$
sum	keine	keine	$\surd$	Literal	$\times$
Inter__0	keine	keine	$\surd$	Synonym	count(?sum)

Tabelle 6.3.: CV-Informationen von GroupBy.sparql

Wie zu sehen ist, wird eine weitere interne CV namens Inter__0 erstellt, die die Aggregationsfunktion ersetzt. Dieses Verfahren wird bereits von Jena eingeführt.

Die Erstellung der InformationSets läuft wie bisher besprochen ab:

- **BGP4**

  Die Auswertung des BGPs gestaltet sich in diesem Beispiel relativ einfach, da es mit Absicht simpel gehalten ist. Das Tripel ?x a : $City$ schränkt die Klassen der CV x auf City ein, damit keine Union-Unterabfrage erstellt werden muss. Die Tripel ?x : $longitude$?lo und ?x : $latitude$?la geben der CV zwei einfache Properties mit, die kein Join erfordern und bis zu diesem Punkt auch noch nicht selektiert werden müssen. Das InformationSet für das BGP4 bleibt daher zunächst leer.

- **Group3**

  Aufgabe des Group3-Knotens ist es, die Gruppierung über die CV sum, die eine Ersetzung für die Konstruktion ?lo+?la darstellt, zu etablieren. Dafür werden die Spalten der CV lo und la mit einem RequireSet von dem BGP4 angefordert. Dieser findet beide CV in seiner lokalen Liste und überprüft mit dem MD, welche Tabellen und Spalten benötigt werden. Diese gibt als Ergebnis einmal City.longitude und einmal City.latitude zurück. Das BGP4 fügt die City-Tabelle mit dem Variablennamen x_City und die beiden Spalten mit den Variablennamen City_lo und City_la seinem InfoSet hinzu. Die beiden Variablennamen der Spalten werden als Antwort auf das RequireSet an Group3 zurückgegeben und werden in die entsprechenden Plätze in der Group By-Klausel für die CV sum eingesetzt. Die CV sum speichert als seinen Variablennamen das komplette Konstrukt mit dem er ersetzt werden soll.

- **Filter2**

  Der Filter speichert ein Complex2Condition-Objekt, das zwei SimpleConditions und den Vergleichsoperator $>$ enthält. In dem ersten SimpleCondition befindet sich die interne CV Inter__0, während die zweite SimpleCondition das Literal 1enthält. Über die explore-Funktion wird die Complex2Condition betrachtet und für die CV Inter__0, die entsprechende Funktion Count mit der Ersetzung von sum eingetragen. Der entstandene Ausdruck „Count((City_lo + City_la)) > 1“ wird im Condition-Block des InformationSets abgelegt.

- **Projection1**

  Den Projection1-Knoten wurde durch die Anfrage mitgeteilt, dass nur die CV sum ausgegeben werden soll. Folglich stellt er ein RequireSet mit der CV sum an Filter2.

Dieser kennt den Variablennamen von sum bereits und gibt diesen als Antwort zurück. Dadurch wird die Ersetzung von sum, also sein Variablenname, in das InformationSet als Select-Ausdruck hinterlegt. Dabei wird ihm der neue Variablenname sum zugeteilt, da die Projection die Ergebnis-Spalten nach den ursprünglich angeforderten Variablen benennt, damit der User die Spalten richtig zuordnen kann.

Bei der folgenden Umwandlung der InformationSets in die SQL-Anfrage gibt es nur eine kleine Besonderheit. Da der Filter-Knoten direkt über einem Group By-Knoten steht und folglich aus einer Having-Klausel entstanden ist, wird der Condition-Abschnitt des Filter-Knotens nicht als Where-Abschnitt ausgegeben und der Group By-Knoten als Unteranfrage im From, sondern korrekterweise die beiden Knoten kombiniert und die Condition als Having unter dem Group By platziert. Die endgültige SQL-Anfrage sieht wie in Abb. 6.8 aus.

```
-------SQL Query-------
Select (City_lo + City_la) as sum
From (

    Select x_City.longitude as City_lo, x_City.latitude as City_la
    From City x_City
    )

GROUP BY (City_lo + City_la)
HAVING ( Count((City_lo + City_la)) > 5)
```

Abbildung 6.8.: Erstellte Anfrage durch GroupBy.sparql

6.2.3. Symmetrische reifizierte Tabellen

```
---------SPARQL Anfrage---------
SELECT ?x ?popx ?y ?popy ?b
WHERE {

    ?b a :Border.
    ?x a :Country.
    ?y a :Country.
    ?b :bordering ?y .
    ?b :bordering ?x .
    ?x :population ?popx .
    ?y :population ?popy

}
```

Abbildung 6.9.: test-symreified.sparql

Anders als in Abb.4.13 werden in dieser Anfrage noch weitere Attribute aus den Tabellen, die mit der reifizierten Tabelle zusammenhängen, angefragt. Dies soll dazu führen, dass

zusätzlich zu der reifizierten Tabelle noch weitere Tabellen selektiert werden müssen, die dann mit jener gejoint werden. Zeitgleich ist es eine gute Demonstration weshalb Tabellen mit der zugehörigen CV kodiert werden müssen, damit mehrfach verwendete Tabellen unterschieden werden können. Die Konvertierung läuft größtenteils ab wie in den bisherigen Beispielen beschrieben, weshalb nur noch auf die Unterschiede eingegangen wird.

CV	Klasse	Known Properties	is Safe	Type
b	Border	bordering x, bordering y	√	Resource
x	Country	hasBorder b, population popx	√	Resource
y	Country	hasBorder b, population popy	√	Resource
popx	Attr. von x	-	√	Literal
popy	Attr. von y	-	√	Literal

Tabelle 6.4.: CV-Informationen von test-symreified.sparql

Wenn für die CV b die Tabellenspalte für eine der beiden bordering-Properties im MD nachgeschlagen wird, so werden zwei Einträge zurückgegeben, nämlich Tabelle: Border, Spalte: Country1 und Tabelle: Border, Spalte: Country2. Dies indiziert, dass es sich bei der Tabelle Border um eine SymmReified-Tabelle handeln muss. Für solche Fälle kann eine CV, die als Objekt in einem solchen Tripel verwendet wird, speichern, auf welche der beiden Einträge sie verweist. So wird der CV y die Spalte Country1 zugewiesen, da sie zuerst behandelt wird. Verlangt ein RequireSet nun nach der CV x, so werden ebenfalls beide MD-Einträge gefunden und es wird überprüft ob die CV des Subjekts (hier b) bereits eine andere Known Property besitzt, deren Property übereinstimmt, aber die CV dazu verschieden ist. In diesem Beispiel wird dann die CV y gefunden, die bereits auf die Spalte Country1 verweist, wodurch für die CV x klar wird, dass sie auf die Spalte Country2 verweisen muss.

Nun sind die CV x und y keine Attribute, sondern besitzen ebenfalls eine Klasse und eine Property, die ausgegeben werden sollen. Es ist klar, dass jeweils die Tabelle Country und von dieser die Spalte Population benötigt wird. Normalerweise werden Duplikate von Tabellen, wie in diesem Fall Country nicht redundant in das InformationSet aufgenommen. Da sie aber von zwei verschieden CV abhängig sind, wird deutlich dass sie keine Duplikate sind und sie werden beide verwendet. Hier ergibt sich auch die Namensgebungskonvention aus CV-Name_Tabelle, da sie für SQL unterscheidbar sein müssen.

Wie in der erstellten Anfrage zu sehen ist, werden durch die oben beschriebenen Zuweisungen von x und y zu unterschiedlichen Spalten der Border-Tabelle ebenfalls die Join-Bedingungen korrekt auf diese Spalten angewandt. So wird eine Country-Tabelle mit Border.Country1 und die andere mit Border.Country2 gejoint.

```
-------SQL Query-------
Select

    x_Country.URI as Result_x,
    x_Country.population as Result_popx,
    y_Country.URI as Result_y,
    y_Country.population as Result_popy,
    b_Border.URI as Result_b

From

    Border b_Border, Country y_Country,
    Country x_Country

Where (b_Border.Country1 = y_Country.URI)
  and (b_Border.Country2 = x_Country.URI)
```

Abbildung 6.10.: Erstellte Anfrage durch test-symreified.sparql

6.2.4. Inverse Properties

Das Problem von inversen Properties wurde im Kapitel 4.3.5 (s. auf Seite 43) schon einmal aufgegriffen. In diesem Beispiel wird die Property flowsThrough verwendet, um für jeden Fluss herauszufinden durch welchen Seen er fließt.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?RN ?LN
WHERE  {

    ?R a :River;
       :name ?RN;
       :flowsThrough ?L .
    ?L :name ?LN
    }
```

Abbildung 6.11.: test-locatedinwaterinv.sparql

Der Syntaxbaum und die Eigenschaften der CVs sind selbstverständlich und werden daher nicht mehr genauer dargestellt. Die einzige Hürde stellt die Auswertung des Tripels ?R :flowsThrough ?L dar.

Durch die Anfrage „Select Tablename t, lookupattr a, inv i From md Where class = 'River' and property = 'flowsThrough' „ erhält man die Antwort in Tabelle 6.5. Lake ist keine NM-Tabelle, da die CV L aber kein Attribut ist, müssen die Tabellen von R mit den Tabellen von L gejoint werden.

Tablename	lookupattr	Inv
Lake	flowsThrough_Inv	true

Tabelle 6.5.: MD-Anfrage für die flowsThrough-Property

Der Join läuft wie ein normaler Join ab, mit der Ausnahme, dass die Tabellen des Subjekts und des Objekts durch die Inversität getauscht werden. So erhält man die Joincondition (L_Lake.flowsThrough_Inv = R_River.URI). Hierbei ist darauf zu achten, dass die hinzugefügte Tabelle Lake auch der Objekt-CV L anstatt wie üblich der Subjekt-CV zugeordnet wird. Die resultierende Anfrage ist in der folgender Abbildung zu betrachten.

```
----------SQL Anfrage---------
Select R_River.name as Result_RN, L_Lake.URI as Result_L
From River R_River, Lake L_Lake
Where (L_Lake.flowsThrough_Inv = R_River.URI)
```

Abbildung 6.12.: Erstellte SQL-Anfrage durch test-locatedinwaterinv.sparql

6.2.5. Join mit null Value

Die vermutlich umfangreichste Sonderbehandlung betrifft den in SQL außergewöhnlichen Join über Null-Werte, dessen Hintergrund im Kapitel 4.3.6 (s. auf Seite 43) bereits erläutert wurde. Es sollte noch einmal darauf hingewiesen werden, dass in SPARQL-Anfragen nur Null-Werte in Verbindung mit einem Optional-Operator entstehen können. Für einen Join bei dem beide Parameter der Joincondition Null-Werte beinhalten können, werden also folglich zwei Optional-Abschnitte benötigt.

An dieser Stelle wird kurz ein Beispiel zum Vergleich angeschnitten, das keinen Join mit Null-Werten erzeugt.

```
----------SPARQL Anfrage---------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?x ?xn ?l ?r ?y
WHERE {
    ?x :name ?xn.
    Optional{
        ?x :locatedAt ?l .
        Optional{
            ?l :hasSource ?r
        }
    }
}
```

Abbildung 6.13.: Anfrage mit zwei Optional ohne Join über Null-Werte

Wie zu erkennen ist handelt es sich bei diesem Beispiel um ein geschachteltes Optional. Gesucht werden alle Objekte mit einem Namen für die, falls vorhanden, ein Gewässer gebunden wird an dem sie liegen. Falls so ein Gewässer existiert, soll wenn möglich die Quelle dieses Gewässers ausgegeben werden. Wie man an dieser Beschreibung schon

merkt, wird das innere Optional nur versucht zu binden sofern das Äußere ein Ergebnis liefert. Dies entspricht auch in SQL einem geschachtelten Left Outer Join und wird daher auch als solches umgesetzt.

Im Gegensatz dazu zeigt Abbildung 6.14 ein Beispiel bei dem über Null-Werte gejoint wird. Dieses Verhalten wird durch die CV *W* erzeugt, die in beiden, wohl bemerkt nicht geschachtelten, Optional-Abschnitten und ansonsten nirgendwo vorkommt. Dadurch kann es vorkommen, dass die Variable entweder von dem ersten Optional-Abschnitt gebunden wird, aber nicht vom Zweiten oder andersherum. Genauso gut können aber auch beide Tripel gleichzeitig ein Ergebnis an die Variable binden, was dann wie bei einem normalen Join die Gleichheit dieser Ergebnisse bedingt.

```
-------SPARQL Anfrage-------
PREFIX : <http://xmlns.com/foaf/0.1/>
SELECT ?C1 ?W ?C2
WHERE { {

    ?C a :Country ;
    :carCode ?CC;
    :hasCity ?C1 .
    FILTER (?CC = 'D')
    OPTIONAL {
        ?C1 :locatedAt ?W }
    }.
    OPTIONAL {
        ?C :hasCity ?C2 .
        ?C2 :locatedAt ?W .
        FILTER (?C1 != ?C2) }
    }
```

Abbildung 6.14.: test-optional-mit-nulls.sparql

Wie in dem Syntaxbaum zu sehen ist unterscheidet sich die Struktur dieser Anfrage von dem vorherigen Beispiel nur in der Position des zweiten Optional-Knotens. Dieser würde sich für das obige Beispiel auf der rechten Seite anstatt der linken befinden. Diese Eigenschaft wird daher benutzt um die oben beschriebenen Fälle zu filtern.

Auf die Auswertung alle anderen Tripel und Knoten wird nicht näher eingegangen, da sie behandelt werden wie es in den vorherigen Kapiteln beschrieben wurde. Der interessante Punkt liegt bei der Erstellung

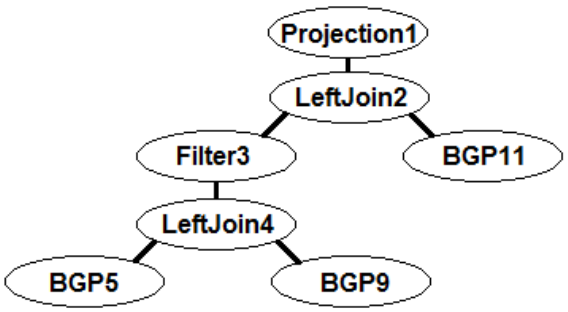

Abbildung 6.15.: Syntaxbaum von test-optional-mit-nulls.sparql

der Joincondition des LeftJoin2-Knotens. Über die findCases()-Methode werden alle Variablen, die in beiden Kindern vorkommen, auf die Eigenschaft unsafe überprüft, die falls true mögliche Null-Werte indiziert. Ist die Variable in beiden Kindern als unsafe markiert und verändert sich dies auch nicht bis zur Projection (überprüfbar durch die FullCVList), so handelt es sich bei dieser um einen Kandidaten bei dem über Null-Werte gejoined werden könnte.

Der Variablenname, der von dem linken Kind hochgereicht wird (Filter3.W_URI), wird als Variablenname für die CV w eingetragen, während der Variablenname von dem rechten Kind (BGP11.W_URI) als AlternativeColumn der CV abgespeichert wird. Für die Joincondition werden vier Sätze an Complex2Conditions angelegt:

1. Deckt den Fall ab, falls beide Kinder ein Ergebnis gebunden haben. Dann werden beide Spalten auf Gleichheit überprüft. ⇒(Filter3.W_URI = BGP11.W_URI)

2. Deckt den Fall ab, falls das linke Kind kein Ergebnis liefert, aber das Rechte schon. ⇒(Filter3.W_URI is null) and (BGP11.W_URI is not null)

3. Deckt den Fall ab, falls das rechte Kind kein Ergebnis liefert, aber das Linke schon. ⇒(Filter3.W_URI is not null) and (BGP11.W_URI is null)

4. Deckt den Fall ab, dass keins der Kinder ein Ergebnis liefert. ⇒ (Filter3.W_URI is null) and (BGP11.W_URI is null)

Alle vier Bedingungen werden mit dem OR-Operator verbunden und als große Complex2Condition der bisherigen Joincondition mit einem AND-Operator hinzugefügt.

Wird später mit einem RequireSet die CV w von LeftJoin2 angefordert, so erfordert diese einen Einsatz der Case-Funktion im Select. Über diese Funktion wird geregelt, dass die AlternativeColumn benutzt werden soll, falls die normale Spalte einen Null-Wert beinhaltet. In SQL wird dies wie folgt umgesetzt: CASE WHEN Filter3.W_URI is null THEN BGP11.W_URI ELSE Filter3.W_URI END as W_URI. Besitzen sowohl Filter3.W_URI als auch BGP11.W_URI einen Wert, so reicht es Filter3.W_URI auszugeben, da sie laut Bedingung in diesem Fall sowieso gleich sein müssen.

Die komplette konvertierte Anfrage ist in der nächsten Abbildung zu sehen. Die wichtigen Stellen für dieses Beispiel sind rot markiert. Zeitgleich wird hier deutlich, dass eine relativ kurze Anfrage in SPARQL eine große SQL-Anfrage nach sich ziehen kann. Dies gibt einen Eindruck darauf, dass komplexere Anforderungen in SPARQL leichter umzusetzen sind.

```
-------SQL Query-------
Select

      Filter3.City_C1_URI as Result_C1,
      CASE WHEN Filter3.W_URI is null
      THEN BGP11.W_URI
      ELSE Filter3.W_URI
      END as Result_W,
      BGP11.City_C2_URI as Result_C2

From (Select BGP5.Country_CC as Country_CC,

      BGP5.Country_C_URI as Country_C_URI,
      BGP9.W_URI as W_URI,
      BGP5.City_C1_URI as City_C1_URI
      From (Select C1_City.URI as City_C1_URI,
                   C_Country.carCode as Country_CC,
                   C_Country.URI as Country_C_URI
            From cityIn C_cityIn,
                 Country C_Country,
                 City C1_City
            Where
             (C_Country.URI = C_cityIn.AdministrativeArea)
             and (C1_City.URI = C_cityIn.City)) BGP5
          Left Outer Join (
              Select C1_City.URI as City_C1_URI,
              W_Union.URI as W_URI
              From locatedAt C1_locatedAt,
                   City C1_City, ( (Select URI URI from Lake)
                                   Union (Select URI URI from Sea)
                                   Union (Select URI URI from River)
                                 ) W_Union
              Where (C1_City.URI = C1_locatedAt.City)
              and (W_Union.URI = C1_locatedAt.Water)) BGP9
          On (BGP5.City_C1_URI = BGP9.City_C1_URI)
          Where (BGP5.Country_CC = 'D')) Filter3

Left Outer Join (

      Select C_Country.URI as Country_C_URI,
             W_Union.URI as W_URI,
               C2_City.URI as City_C2_URI
      From cityIn C_cityIn,
           Country C_Country,
             City C2_City,
             locatedAt C2_locatedAt,( (Select URI URI from Lake)

                Union (Select URI URI from Sea)

                  Union (Select URI URI from River)
                                   ) W_Union
      Where (C_Country.URI = C_cityIn.AdministrativeArea)
          and (C2_City.URI = C_cityIn.City)
          and (C2_City.URI = C2_locatedAt.City)
          and (W_Union.URI = C2_locatedAt.Water)) BGP11

On ((Filter3.Country_C_URI = BGP11.Country_C_URI)

      and ((Filter3.W_URI = BGP11.W_URI)
      or (((Filter3.W_URI is null)
      and (BGP11.W_URI is not null))
      or ((Filter3.W_URI is null)
      and (BGP11.W_URI is not null))
      or ((Filter3.W_URI is null)
      and (BGP11.W_URI is null))
      )
      )
)
```


Abbildung 6.16.: SQL Query von test-optional-mit-nulls.sparql

7. Zusammenfassung und Ausblick

Zusammen sind RDF2SQL und Query Converter in der Lage aus einem RDF-Datenbestand, eine SQL-Datenbank mit den erweiterten Funktionalitäten von RDF zu bauen und entsprechende Anfragen äquivalent umzuformen. Die generelle Machbarkeit einer RDF zu SQL Umformung wurde zwar bereits anderweitig gezeigt, allerdings konnte gezeigt werden, dass sich diese Konvertierung durch die Theorie von [10] praktisch umsetzen lässt.

Außerdem gibt es ein Webinterface für den Query Converter, damit die Funktionsweise einfach und schnell getestet werden kann.

Der Query Converter hat noch nicht seine maximal möglichen Funktionen, so könnten z.B. Existenzquantoren eingeführt werden, wobei es sich hier lohnt zu warten, bis Jena diese im vollen Umfang implementiert.

Anmerkung

Nur das Programm *Query Converter* ist im eigentlich Rahmen der Bachelorarbeit entstanden, das Programm *RDF2SQL* ist im vorher gehenden Praktikum von uns angefertigt worden, nicht in der Bachelorarbeit selbst. Einige Dinge werden hier direkt aus dem Praktikumsbericht übernommen, ohne explizit gekennzeichnet zu werden. Für die Implementierung verweisen wir ebenfalls auf den Praktikumsbericht [14].

Literaturverzeichnis

[1] Central Intelligence Agency. The World Factbook 2013-14. Website. Available online at https://www.cia.gov/library/publications/the-world-factbook/index.html; visited on March 13th 2014.

[2] Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. Technical report, Web-based Systems Group, Freie Universität Berlin, 2009. Available from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.161.8030&rep=rep1&type=pdf; visited on March 11th 2014.

[3] Francois Bry, Michael Kraus, Dan Olteanu, and Sebastian Schaffert. Aktuelles Schlagwort "Semi-strukturierte Daten". Technical report, Universität München, Institut für Informatik, 2001. Available from http://www.en.pms.ifi.lmu.de/publications/PMS-FB/PMS-FB-2001-9.pdf; visited on February 22th 2014.

[4] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. Technical report, Massachusetts Institute of Technology, 1974. Available from http://dl.acm.org/citation.cfm?id=811515; visited on March 2nd 2014.

[5] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. Technical report, Massachusetts Institute of Technology, 1976. Available from http://www.colonese.it/00-Sw-Engineering_Articoli/Entity-Relationship%20Model,%20Peter%20Chen,%201976.pdf; visited on February 13th 2014.

[6] Heinz-Dieter Ebbinghaus. *EinfÃŒhrung in die Mengenlehre*. Wiss. Buchges., 1977.

[7] The Apache Software Foundation. Jena homepage. Website. Available online at http://jena.apache.org; visited on January 28th 2014.

[8] The Apache Software Foundation. SPARQL Tutorial. Website. Available online at http://jena.apache.org/tutorials/sparql.html; visited on March 11th 2014.

[9] RDF Working Group. RDF Homepage. Website. Available online at http://www.w3.org/RDF; visited on January 25th 2014.

[10] Thomas Hornung and Wolfgang May. Efficient, Schema-Aware Storage of RDF. Technical report, Universität Freiburg, Institut für Informatik, Universität Göttingen, Institut für Informatik, 2012.

[11] Stefan Kokkelink, Judith Plümer, Hartmut Polzer, and Roland Schwänzl. Speicherung von RDF in Datenbanken. Technical report, Institut für wissenschaftliche Information e.V., 2000. Available from http://www.iwi-iuk.org/seminarNotes/1/RDFdb.pdf; visited on March 12th 2014.

[12] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from http://dbis.informatik.uni-goettingen.de/Mondial.

[13] Oracle. SQL Developer Homepage. Website. Available online at http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html; visited on February 1st 2014.

[14] Lars Runge and Sebastian Schrage. Forschungspraktikumsbericht RDF-to-SQL-Konverter. Technical report, Universität Göttingen, Institut für Informatik, 2013.

[15] W3C. W3C Recommendation. Website. Available online at http://www.w3.org/TR/rdf-sparql-query; visited on March 2nd 2014.

A. Klassendiagramme

Abbildung A.1.: Klassendiagramm der CV

Abbildung A.2.: Klassendiagramm des Node Packages

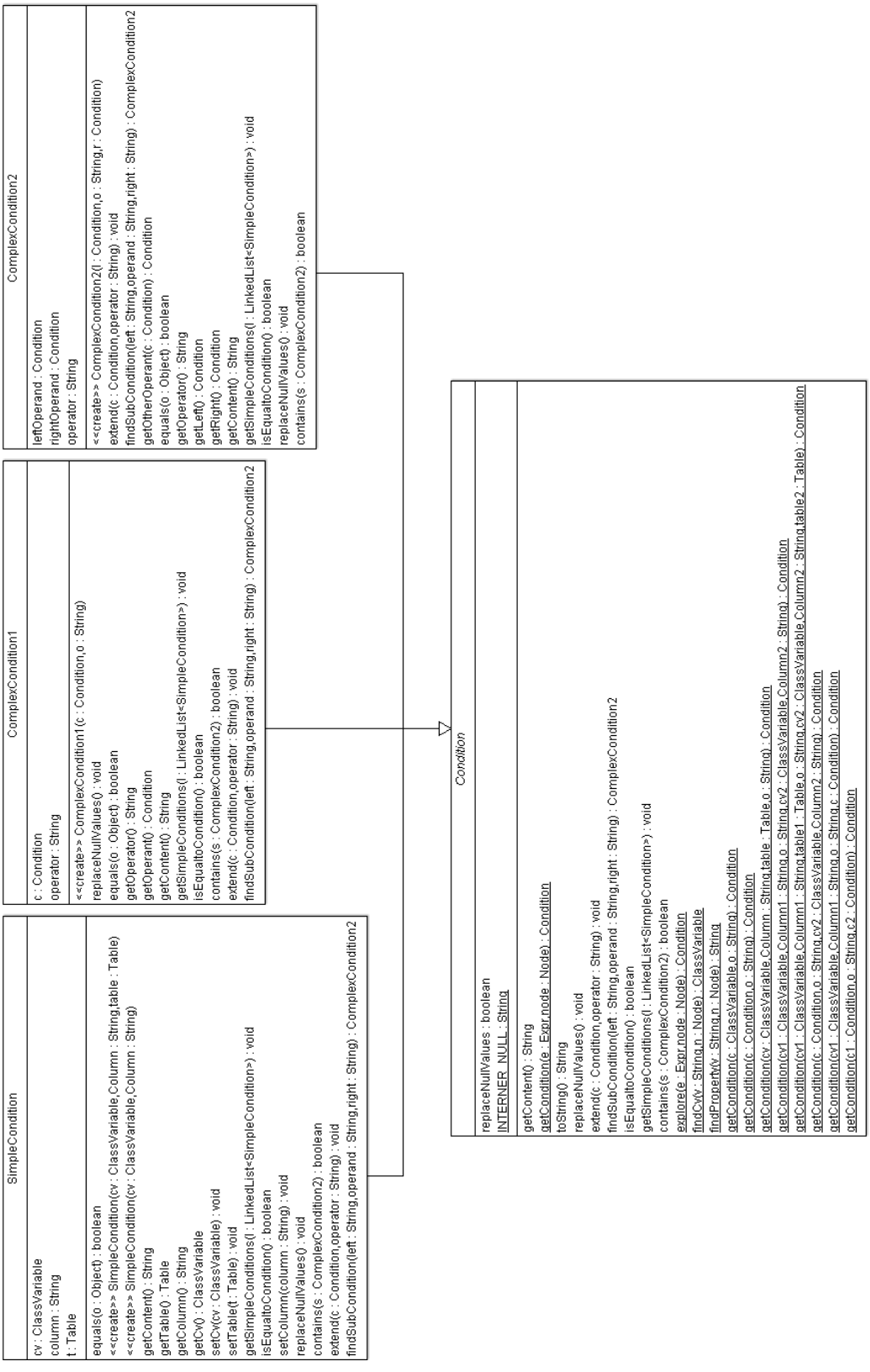

Abbildung A.3.: Klassendiagramm der Conditions

B. Aufteilung

Thema	Autor
1. Einführung	-
1.1	Lars Runge, Sebastian Schrage
1.2.1	Sebastian Schrage
1.2.2	Lars Runge
2. Grundlagen	-
2.1 - 2.2	Sebastian Schrage
2.3 - 2.3.1	Lars Runge
2.4.1	Sebastian Schrage
2.4.2	Lars Runge
2.4.3	Sebastian Schrage
2.4.4 - 2.4.6	Lars Runge
2.4.7 - 2.4.8	Sebastian Schrage
3. RDF2SQL	-
3.1.1	Lars Runge
3.1.2	Sebastian Schrage
3.1.3 - 3.2.2	Lars Runge
3.2.3-3.2.4	Sebastian Schrage
4. Query Converter	-
4.1	Sebastian Schrage
4.2.1 - 4.2.2	Sebastian Schrage
4.2.3 - 4.2.7	Lars Runge
4.2.8	Sebastian Schrage
4.3.1	Lars Runge
4.3.2 - 4.3.5	Sebastian Schrage
4.3.6	Lars Runge
5. Implementierung	
5.1	Sebastian Schrage
5.2.1	Sebastian Schrage
5.2.2 - 5.2.3	Lars Runge
5.2.4.1	Sebastian Schrage
5.2.4.2 - 5.2.4.7	Lars Runge
5.2.4.8	Sebastian Schrage
5.2.5.1 - 5.2.5.2	Lars Runge
5.2.5.3	Sebastian Schrage
6. Anwendung	-
6.1	Sebastian Schrage
6.2.1 - 6.2.3	Lars Runge
6.2.4	Sebastian Schrage
6.2.5	Lars Runge
7. Zusammenfassung	Lars Runge, Sebastian Schrage
Klassendiagramme	Sebastian Schrage

Tabelle B.1.: Autoren zu den Einzelbereichen