ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG
INSTITUT FÜR INFORMATIK

RDFPath

Verteilte Analyse von RDF-Graphen

MASTERARBEIT

eingereicht von

Martin Przyjaciel-Zablocki

am

Lehrstuhl für Datenbanken und Informationssysteme
Dozent: Prof. Dr. Georg Lausen
Betreuer: Dipl.-Inf. Thomas Hornung

3. September 2010

Kurzfassung

Gegenstand dieser Arbeit ist die Frage, ob eine Analyse von *RDF-Graphen* unter Verwendung des *Apache Hadoop-Frameworks* effizient durchgeführt werden kann. Da im Rahmen der Analyse insbesondere Pfade in RDF-Graphen betrachtet werden sollen, wird zunächst eine Pfadanfragesprache eingeführt, die *RDFPath* benannt wird. Darauf aufbauend wird die entwickelte Implementierung vorgestellt, die Pfadanfragen auf Grundlage von *MapReduce* verteilt auswertet. Die wesentlichen Komponenten dieses Systems sind zum einen *Reduce-Side-Joins*, die eine zentrale Rolle im Auswertungsprozess einnehmen und zum anderen der *RDFPath-Store*. Letzteres beschreibt ein Datenkonzept, das durch die vertikale Partitionierung von *RDF-Dokumenten* eine effiziente Auswertung begünstigt. Die Implementierung wird abschließend sowohl auf synthetisch generierten als auch auf realen Daten evaluiert. Hierbei wird gezeigt, dass eine verteilte Analyse von RDF-Graphen unter Verwendung von RDFPath als Analysewerkzeug sinnvoll durchgeführt werden kann. Sowohl die lineare Skalierung vieler Anfragen, als auch die unterschiedlichen Fragestellungen, die mit der eigens entworfenen Pfadanfragesprache betrachtet werden konnten, werden im Rahmen dieser Arbeit positiv bewertet.

Schlagwörter: RDFPath, MapReduce, Hadoop, HDFS, Resource Description Framework, RDF-Graph, Pfadanfrage, RDFPath-Store, RDF Triple Store, SP²Bench, Last.fm, SPARQL

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Erzeugung von neuen Daten hat in den letzten Jahren rasant zugenommen. Google-Chef Eric Schmidt hat in diesem Zusammenhang auf der Techonomy-Konferenz 2010[1] in Kalifornien einen interessanten Vergleich geführt, der verdeutlichen sollte, wie stark sich die Gesellschaft im Informationszeitalter verändert hat. Demnach produzieren die Menschen alle zwei Tage 5 Exabytes an Daten, was den gesamten Datenbeständen entspräche, die bis zum Jahre 2003 generiert wurden [Schmidt, et al., 2010].

Auch wenn diese Zahlen gewiss schwierig zu belegen sind, so zeigen sie dennoch eine Entwicklung auf, mit der sich heutzutage viele Unternehmen auseinandersetzen müssen. So generiert zum Beispiel die New York Stock-Exchange jeden Tag ein Terabyte an Handelsdaten [White, 2009].

Ebenso bemerkenswert sind auch die Zahlen, die von Facebook veröffentlicht wurden. Gemäß den aktuellen Pressemitteilungen[2] wird das soziale Netzwerk von mehr als 500 Millionen Nutzern aktiv verwendet, die mit mehr als 900 Millionen unterschiedlichen Objekten interagieren können. Die hieraus resultierenden Datenmengen stellen Forschung wie Wirtschaft vor neue Herausforderungen. Denn je größer die Datenmengen werden, umso wichtiger wird es sein, effiziente Möglichkeiten für die Verarbeitung und Analyse stetig wachsender Datenbestände zu finden.

Immer häufiger werden solche Daten in eine Repräsentationsform gebracht, die es Maschinen ermöglicht, sie ohne den Menschen zu interpretieren und automatisch weiterzuverarbeiten. Im Kontext des *Semantischen Webs* spricht man hier gerne von der Vision, alle Informationen dieser Welt mit ihrer Bedeutung zu verknüpfen, sodass sie auch maschinell verstanden werden können. Hierzu ist allerdings eine eindeutige und universelle Repräsentationsform erforderlich, welche beliebige Daten strukturiert darstellen kann und es erlaubt, sie mit anderen verfügbaren Informationen zu verbinden. Einer immer größer werdenden Beliebtheit erfreut sich dabei das sogenannte *Resource Description Framework* (RDF), eine formale Sprache zur Modellierung von Metainformationen. Dabei können Informationen, die unter Verwendung dieser Sprache beschrieben worden sind, als *RDF-Graphen* visualisiert werden [Champin, 2001].

---

[1] http://techonomy.com/

[2] http://www.facebook.com/press/info.php?statistics

Auch wenn sich mit dem Resource Description Framework Datenmengen strukturiert darzustellen lassen, bleibt die Frage offen, wie man diese Tera- bis Petabyte großen RDF-Graphen nicht nur abspeichern, sondern auch effizient analysieren kann. Viele bisherige Bestrebungen setzen dabei auf Hochleistungsrechner oder die individuelle Entwicklung *verteilter Systeme*, die sowohl hardware- als auch softwareseitig für die Unternehmen mit einem sehr hohen Kostenaufwand verbunden sind [Dean, et al., 2004].

Eine vielversprechende Lösung hierfür kommt aus dem Bereich *des Distributed Computing*. Mit *MapReduce* hat Google ein einfaches jedoch leistungsfähiges Konzept entwickelt, welches die Verteilung der Daten und die Parallelisierung der Berechnungen automatisch übernimmt [Dean, et al., 2004]. Entwickler, die ihre häufig sehr datenintensiven Probleme unter Verwendung des *MapReduce-Programmiermodells* lösen, können sich dabei voll und ganz auf die eigentliche Fragestellung konzentrieren und müssen sich keine Gedanken mehr über die Synchronisation der Daten, Netzwerkprotokolle oder die Ausfallsicherheit der Server machen. Allerdings steht Google's MapReduce nur unternehmensintern zur Verfügung.

Daher wurde mit dem quelloffenen *Apache Hadoop-Framework* eine freie Implementierung des MapReduce-Programmiermodells entwickelt, die ähnliche Eigenschaften aufweist [White, 2009]. Der Erfolg dieses Frameworks zeigt sich in den zahlreichen, namhaften Unternehmen[3] wie beispielsweise Adobe, Twitter oder IBM, die besonders datenintensive Probleme zu lösen haben. Mittlerweile verfügen sie über eigene *Hadoop-Cluster*, die in manchen Fällen sogar mehrere tausend Rechner umfassen. Unternehmen, die keinen eigenen Hadoop-Cluster benötigen, bietet Amazon unter dem Namen *Amazon Elastic MapReduce*[4] Rechenkapazitäten in einem flexiblen Hadoop-Framework zur Miete an.

Gegenstand dieser Arbeit ist daher die Frage, ob eine Analyse von großen RDF-Graphen unter Verwendung des Hadoop MapReduce-Frameworks effizient durchgeführt werden kann. Die Grundlage der Analyse sollen dabei *Pfade* in RDF-Graphen bilden, die mit einer entsprechenden *Anfragesprache* ausgedrückt werden können.

[3] Weitere Beispiele: http://wiki.apache.org/hadoop/PoweredBy

[4] Amazon Elastic MapReduce: http://aws.amazon.com/de/elasticmapreduce/

1.1 Zielsetzung

Um die zuvor aufgestellte Fragestellung zu beantworten, ob und wie eine verteilte Analyse von RDF-Graphen mit MapReduce möglich ist, musste im Rahmen dieser Arbeit ein System entwickelt werden, das solche Analysen durchführen kann. Ziel war also im ersten Schritt eine Implementierung auf Grundlage des Hadoop MapReduce Frameworks zu entwerfen, die folgenden Funktionsumfang aufweist:

- RDF-Daten sollten eingelesen werden können.
- Ein internes Datenkonzept sollte spezifiziert werden.
- Die Speicherung und Verteilung der RDF-Daten in Hadoop sollte im Hinblick auf eine möglichst effiziente Ausführung von Pfadanfragen erfolgen.
- Die Analysen sollten unter Verwendung einer Pfadanfragesprache durchgeführt werden können, die im Rahmen dieser Arbeit spezifiziert werden sollte.
- Die Pfadanfragesprache sollte durch eine Übersetzung in MapReduce Aufträge verteilt ausgewertet werden können.
- Die Interaktion mit dem implementierten System sollte über eine textbasierte Eingabeaufforderung erfolgen.

Im zweiten Schritt sollten neben der ursprünglichen Fragestellung auch neue Erkenntnisse über MapReduce und Hadoop im Allgemeinen und der implementierten Strategien im Speziellen gewonnen werden. Hierbei sollten insbesondere die folgenden Aspekte genauer betrachtet werden:

- Ist Hadoop's MapReduce-Framework eine geeignete Grundlage für die verteilte Analyse von RDF-Graphen?
- Was lässt sich über das Skalierungsverhalten der Anfragen aussagen?
- Ab welcher Dateigröße und bis zu welcher Dateigröße ist eine Auswertung auf dem Hadoop-Cluster sinnvoll?
- Welche Flaschenhälse können identifiziert werden?
- Welche Aspekte gilt es bei der Implementierung zukünftiger Fragestellungen, die auf Grundlage von Hadoop's MapReduce-Framework konzipiert werden, zu berücksichtigen?

1.2 Kapitelüberblick

Der Aufbau dieser Ausarbeitung ist in Abbildung 1 grafisch dargestellt. In Kapitel 2 werden die wesentlichen Sprachen und Technologien dargelegt, die im weiteren Verlauf dieser Arbeit aufgegriffen werden. Hierzu zählen das *Resource Description Framework (RDF)* und *Hadoop.* Beim Letzteren wird insbesondere das *MapReduce-Programmiermodell* ausführlich eingeführt. Außerdem wird für das Verständnis einiger Optimierungsansätze auf die *Berkeley Database* eingegangen.

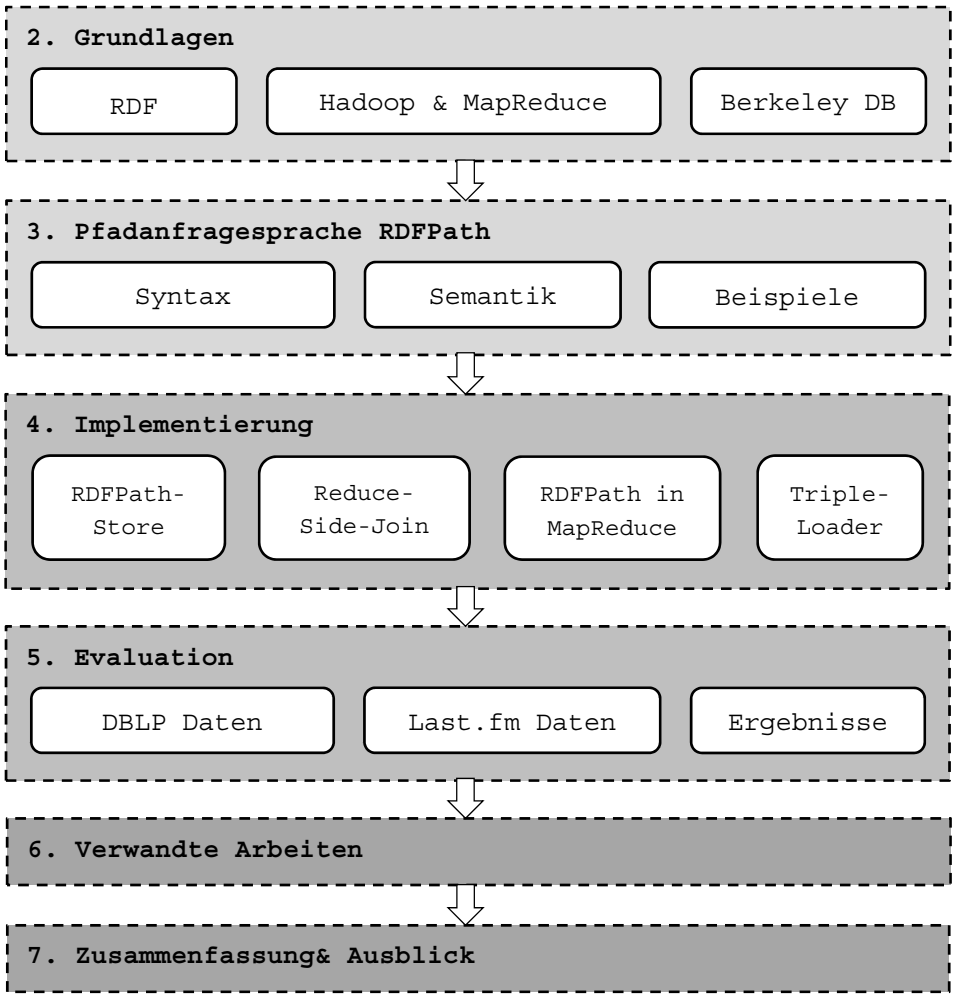


Abbildung 1: Aufbau der Arbeit

Im darauffolgenden Kapitel 3 wird die *Pfadanfragesprache RDFPath* vorgestellt, die ihm Rahmen dieser Arbeit spezifiziert und implementiert wurde. Darauf aufbauend werden in Kapitel 4 die unterschiedlichen Komponenten der Implementierung und ihr Zusammenspiel besprochen. Hierbei wird zunächst auf die Datenspeicherung eingegangen, die unter dem Begriff *RDFPath-Store* zusammengefasst wird. Weiterhin werden die sogenannten *Reduce-Side-Joins* beschrieben, welche die Kernkomponente des Auswertungsprozesses darstellen.

Ferner wird in diesem Kapitel auch die Überführung von RDFPath Anfragen in *MapReduce Aufträge* besprochen und abschließend mit dem *Triple-Loader* eine Komponente vorgestellt, die das Einpflegen von *RDF-Dokumenten* in den RDFPath-Store mit oder ohne *Dictionary Encoding* durchführen kann. In Kapitel 5 wird eine Evaluation der Implementierung vorgestellt, die auf zwei unterschiedlichen Datenquellen – synthetisch generierte und reale Datensätze – durchgeführt wurde. Die Ergebnisse dieser Evaluation werden im letzten Abschnitt dieses Kapitels ausführlich besprochen und analysiert. In Kapitel 6 werden einige verwandte Arbeiten vorgestellt, die im Bereich des *Distributed Computing* ähnliche Fragestellungen zu beantworten versuchen. Die Ausarbeitung wird mit Kapitel 7 abgeschlossen, indem eine kurze Zusammenfassung gegeben wird und mögliche Optimierungs- und Erweiterungsansätze skizziert werden.

2 Grundlagen

In diesem Kapitel werden die unterschiedlichen Themengebiete eingeführt, mit denen sich diese Arbeit auseinandersetzt, indem einige ausgewählte Technologien und Sprachen vorgestellt werden. Die Ausgangssituation stellen, wie schon in der Motivation erläutert, Datenmengen im Tera- bis Petabyte Bereich dar. Um diese Datenmengen sinnvoll maschinell verarbeiten zu können, müssen sie eine eindeutige Struktur beziehungsweise ein definiertes *Schema* aufweisen. Mit einem solchen Schema ist es möglich, beliebigen Daten, wie etwa einer gemessenen Temperatur, einen Kontext zuzuordnen. Dieser könnte beispielsweise aussagen, an welchem Ort und zu welcher Zeit dieser Wert gemessen wurde. Im Bereich des *Semantischen Webs* wird für solche Zwecke das im nächsten Abschnitt beschriebene *Resource Description Framework* vorgeschlagen. Die nächste Herausforderung besteht in der Verwaltung und Analyse der Datenbestände. Eine Lösung hierfür könnte das von Google entworfene *MapReduce-Programmiermodell* beziehungsweise die quelloffenen Implementierung *Apache Hadoop* darstellen. Im Anschluss an die Einführung in Hadoop wird für das Verständnis einiger Optimierungsstrategien die *Berkeley Database* in ihren Grundzügen vorgestellt.

2.1 Resource Description Framework

Das *Resource Description Framework* (RDF) bezeichnet eine vom *World Wide Web Consortium*[5] (W3C) standardisierte Sprache, die zur formalen Beschreibung von *Aussagen* und Informationen über *Ressourcen* konzipiert wurde. Unter einer Ressource versteht man in diesem Kontext eine beliebige *Entität*, die zur Modellierung von Wissen verwendet werden könnte. Sie wird durch einen eindeutigen Bezeichner, auch *Uniform Resource Identifier* (URI) genannt, identifiziert. Dieser Bezeichner besteht aus einer Zeichenfolge, die – analog zur *URL*[6] – ein Dokument im Internet unverwechselbar referenziert, sich jedoch nicht nur auf im Internet verfügbare Inhalte beschränkt. Mit sogenannten *Blank Nodes* ist es außerdem möglich, lokale Ressourcen zu verwenden, die nur innerhalb eines Dokumentes eindeutig sind und dementsprechend auch keine URIs besitzen. Sie dienen im Allgemeinen zur Gruppierung von Informationen.

[5] Gremium zur Standardisierung von World Wide Web Techniken

[6] Uniform Resource Locator: Unterart von URIs, ugs. Internetadresse

Weiterhin können für die Modellierung von Informationen über Ressourcen auch einfache *Literale* verwendet werden. Darunter sind atomare Werte zu verstehen, die im Regelfall Zeichenketten oder Zahlen repräsentieren. Übliche Beispiele hierfür können das Alter oder der Name einer Person sein.

Das Grundelement des *RDF-Modells* stellt das *RDF-Triple* dar. Es besteht aus drei Komponenten: einem *Subjekt*, einem *Prädikat* und einem *Objekt*. Bei einem Triple werden zwei Ressourcen, das Subjekt und das Objekt, über eine dritte Ressource, das Prädikat, miteinander in Verbindung gebracht, um damit eine *Aussage* über eine Ressource auszudrücken. Diese Aussage beschreibt nach [Champin, 2001] folgenden Zusammenhang:

<Subjekt> hat eine Eigenschaft **<Prädikat>** mit dem Wert **<Objekt>**

Während an die Stelle eines *Subjektes* sowohl eine URI als auch ein Blank Node gesetzt werden kann, handelt es sich bei *Prädikaten* immer um URIs. Objekte hingegen dürfen neben URIs und Blank Nodes zusätzlich Literale enthalten.

Für die Repräsentation eines solchen RDF-Modells, das eine Vielzahl von RDF-Triples beinhalten kann, werden mehrere Formate vorschlagen. Das W3C spezifizierte zum einen das *XML-Format*[7] *RDF/XML* und zum anderen das für Menschen lesbarere Format *RDF/Notation 3* (N3) [Manola, et al., 2004]. Im weiteren Verlauf dieser Ausarbeitung wird die Verwendung von RDF-Dokumenten im N3 Format angenommen. Weitere Details, die das *Einlesen* und *Interpretieren* eines solchen Dokumentes betreffen, werden in Abschnitt 4.4 erörtert.

Eine weitere Interpretationsmöglichkeit von RDF-Dokumenten sind *RDF-Graphen*. Die Menge an Triples in einem RDF-Dokument lässt sich auch als ein gerichteter, azyklischer Graph darstellen, wobei die Subjekte und Objekte als Knoten und die Prädikate als Kanten zu verstehen sind [Powers, 2003]. In Abbildung 2 ist ein beispielhaftes RDF-Dokument im N3-Format dargestellt.

```
@prefix pe: <http://example.com/> .
@prefix foaf: <http://xmlns.com/foaf/> .

pe:Peter    foaf:kennt    pe:Simon .
pe:Peter    foaf:kennt    pe:Sarah .
pe:Simon    foaf:kennt    pe:Sarah .
```

Abbildung 2: RDF-Dokument im N3-Format

[7] Extensible Markup Language: http://www.w3.org/XML/

Die ersten zwei Zeilen des RDF-Dokumentes sind sogenannte *Präfixe*, ein Hilfsmittel um lange URIs im gesamten RDF-Dokument kompakter darstellen zu können. Darunter sind drei Triples der Form `(Subjekt, Prädikat, Objekt)` definiert. Der dazugehörige RDF-Graph zu diesem Dokument ist in Abbildung 3 zu sehen. Um die Übersichtlichkeit zu erhöhen, wird im weiteren Verlauf dieser Ausarbeitung auf die Darstellung von Präfixen verzichtet. Das bedeutet, Ressourcen, die man üblicherweise über längere URIs der Form `„http://example.com/Peter“` eindeutig identifiziert, werden hier vereinfacht als `„Peter“` bezeichnet.

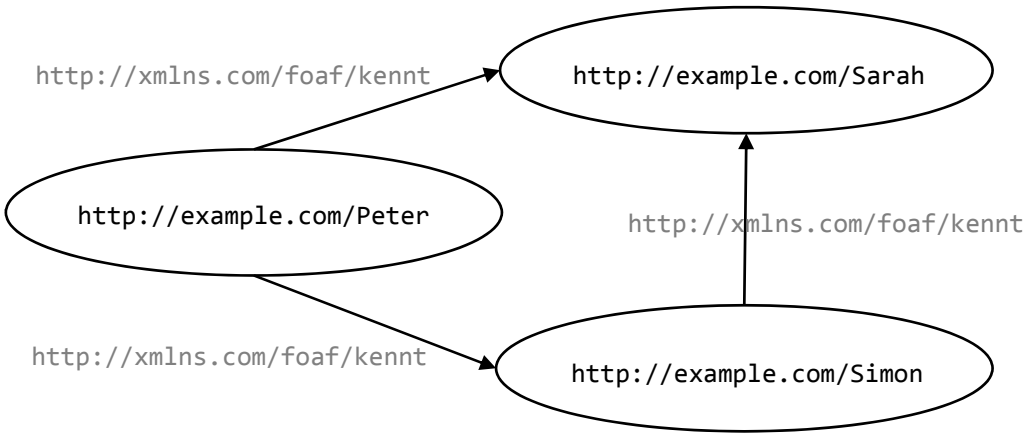


Abbildung 3: RDF-Dokument als RDF-Graph

2.2 Hadoop

Google stand, wie viele andere Firmen heute auch, vor der Aufgabe, immer größer werdende Datenmengen nicht nur speichern, sondern auch verarbeiten zu wollen. Die Auswertung von mehreren Petabyte großen Log-Dateien oder die Generierung eines *invertierten Indexes* [Knuth, 1998] für Googles Internetsuche erforderte die Entwicklung zahlreicher individuell angepasster Systeme [Dean, et al., 2004]. Dies wurde notwendig, da traditionelle Lösungen aus dem Bereich *der relationalen Datenbankmanagementsysteme (DBMS)* den neuen Anforderungen nicht immer gerecht wurden.

Für die Stapelverarbeitung von sehr großen Datenmengen sind die Vorteile eines datenbankbasierten Systems, die sich beispielsweise durch kurze Reaktionszeiten und eine hohe Datenintegrität auszeichnen, nicht länger von Bedeutung. Auch gibt es für viele Aufgabenstellungen keine Notwendigkeit, einzelne Datensätze im Nachhinein zu bearbeiten. Auf Daten soll, nachdem sie einmal abgespeichert wurden, nur noch lesend zugegriffen werden. Das wesentliche

Problem von Datenbankmanagementsystemen bestehen jedoch in ihrer Erweiterbarkeit und Parallelisierbarkeit. So erfordern diese Systeme zum einen häufig teure Hochleistungsrechner[8] und zum anderen skalieren sie ab mehreren hundert Rechnern nicht mehr linear mit der Anzahl zur Verfügung stehender Maschinen [Azza Abouzeid, 2009]. Weiterhin können Daten deutlich schneller verarbeitet werden, als von üblichen magnetischen Speichermedien[9] gelesen werden kann. Demzufolge ist es erforderlich, Lesevorgänge parallel unter Verwendung hunderter Laufwerke auszuführen. Hierzu gibt es bereits etablierte verteilte Systeme wie das *Grid-Computing* oder das *High Performance Computing*. Bei diesen Ansätzen liegt das Hauptaugenmerk jedoch mehr auf der Rechenleistung als auf der Menge der zu verarbeitenden Daten. Diese werden je nach Bedarf zu den entsprechenden Knoten transferiert, was allerdings schon ab mehreren hundert Gigabyte zu einem Problem für das Netzwerk werden kann [White, 2009].

Googles Antwort auf diese Problematik ist ein skalierendes, verteiltes System, welches die Implementierung von datenintensiven Anwendungen erleichtern soll. Entwickler, die auf diese, als *MapReduce* bekannt gewordene, Architektur, aufsetzen, müssen sich beispielsweise keine Gedanken mehr über die Organisation und Synchronisation ihrer Daten machen. Eine hohe Fehlertoleranz, die durch *Replikation* erreicht wird, ermöglicht auch den Einsatz gewöhnlicher *Standard-Hardware*[10] in Rechnerclustern. Diese Aspekte werden durch das im Jahre 2003 in [Ghemawat, et al., 2003] vorgestellte *Google File System* abgedeckt.

Für die automatische Parallelisierung von Berechnungen stellten Entwickler bei Google in [Dean, et al., 2004] das *MapReduce-Programmiermodell* vor. Diese Abstraktionsschicht stellt Programmierern ein einfaches jedoch mächtiges Werkzeug zur Verfügung, um Probleme auf großen Standard-Hardware Rechnerclustern verteilt zu lösen. Hierfür ist lediglich die Implementierung einer *Map-* sowie einer *Reduce-Funktion* erforderlich. Dabei werden die Daten nicht durch das Netzwerk zu der jeweiligen Berechnung verschoben, sondern die Berechnungen finden genau dort statt, wo die Daten gespeichert wurden. Dieses auch als *Data Locality* bekannte Konzept setzt allerdings voraus, dass Daten in kleine Teile aufgeteilt und unabhängig voneinander verarbeitet werden können.

---

[8] Individuell hergestellte Rechner aus leistungsfähiger und fehlertoleranter Spezialhardware

[9] Für die Zukunft ist in diesem Kontext die Entwicklung von Solid State Drives (SSD) zu beachten.

[10] Hardwarekomponenten, die nicht nur für den professionellen Einsatz in Servern hergestellt werden. Ein Beispiel hierfür ist in Kapitel 5.1.1 nachzulesen.

Folglich kann es auch Problemstellungen geben, die mit diesem Programmiermodell nur umständlich zu lösen sind und die Vorteile des Frameworks nicht ausnutzen können. Ein weiterer, wesentlicher Punkt des MapReduce Frameworks betrifft die *Skalierbarkeit* des Clusters. Zusätzliche Auslastung durch größere Datenmengen oder komplexere Anfragen sollte nicht zu Fehlern und Abstürzen führen, sondern lediglich die Ausführungszeit beeinflussen. Hingegen sollten zusätzliche Hardwareressourcen, wie zum Beispiel weitere Rechner, die Belastbarkeit beziehungsweise die maximale Kapazität des Clusters proportional erhöhen, sodass auch der Einsatz von mehreren tausenden Rechnern möglich wird [Dean, et al., 2004].

Bei *Hadoop* handelt es sich um eine quelloffene Implementierung des *Google File Systems* und des *MapReduce Programmiermodells*, die mit den von Google veröffentlichten Eigenschaften übereinstimmt. Google stellt seine eigene Implementierung nur unternehmensintern zur Verfügung, sodass externe Interessenten die quelloffene Variante nutzen müssen. Die Entwicklung von Hadoop wurde dabei maßgeblich von Yahoo! Mitarbeitern als ein Projekt der Apache Software Foundation[11] vorangetrieben. Als Programmiersprache kam hierbei größtenteils Java zum Einsatz, wobei auch Schnittstellen zu weiteren Programmiersprachen, wie C++, Ruby oder Python zur Verfügung gestellt werden. In den letzten Jahren erfreute sich Hadoop einer immer größer werdenden Beliebtheit. Viele namhafte Unternehmen wie beispielsweise Adobe, Twitter, EBay, IBM oder Last.fm haben mittlerweile eigene Hadoop Rechencluster aufgebaut. Für Unternehmen, die keinen eigenen Hadoop Rechencluster benötigen, bietet Amazon unter dem Namen *Amazon Elastic MapReduce* Rechenkapazitäten in einem flexiblen Hadoop-Framework zur Miete an.

2.2.1 Apache Hadoop Projekte

Das *Apache Hadoop Projekt* umfasst mehrere Unterprojekte, die sich mit unterschiedlichen Problemstellungen des Distributed Computing auseinandersetzen. Ein Großteil dieser Projekte wurde in Anlehnung an entsprechende Veröffentlichungen von Google implementiert. In Abbildung 4 aus [White, 2009] sind einige zentrale Unterprojekte zu sehen. Die entsprechenden Äquivalente von Google sind in Klammern aufgeführt.

[11] gemeinnützige Organisation zur Förderung von quelloffenen Softwareprojekten http://www.apache.org/

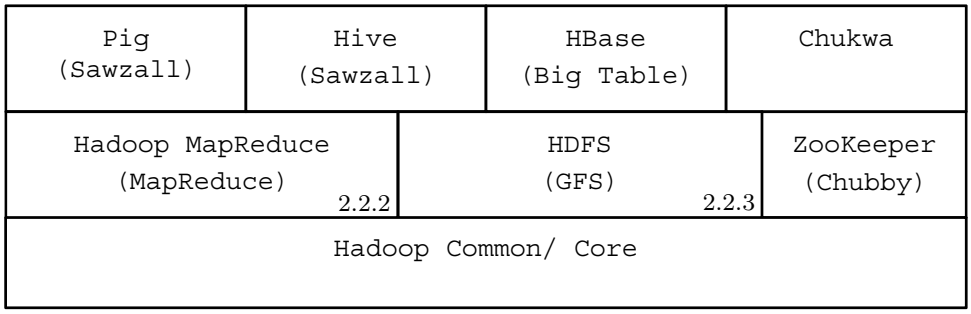


Abbildung 4: Apache Hadoop Projekte [White, 2009]

Im Nachfolgenden werden einige ausgewählte Apache Projekte in ihren Grundzügen erläutert. Die beiden für die Ausarbeitung wesentlichen Bestandteile, zu denen die *Hadoop MapReduce* Implementierung sowie das *Hadoop Distributed Filesystem* (HDFS) zählen, werden im Anschluss an diesen Abschnitt gesondert besprochen.

Pig

Bei *Pig* handelt es sich um eine von Yahoo! Research entwickelte Lösung, um Datenanalysen im MapReduce Framework zu vereinfachen. Anfragen können in der eigens entworfenen Anfragesprache *Pig Latin* geschrieben werden, welche die deklarative Struktur von SQL mit prozeduralen Aspekten einer Programmiersprache vereint. Da die Anfragen automatisch in eine Folge von MapReduce Aufträgen übersetzt werden, brauchen sich Anwender nicht mehr mit der Implementierung von Map- und Reduce-Funktionen zu befassen [Olston, et al., 2008].

Hive

Ähnlich wie *Pig* hat auch diese, von Facebook entwickelte, Lösung zum Ziel, Datenanalysen im MapReduce Framework zu vereinfachen. Hier wird von einem verteilten *Data-Warehouse* gesprochen, wobei als Anfragesprache ein SQL-Dialekt zum Einsatz kommt.

HBase

HBase ermöglicht die verteilte Speicherung von *spaltenorientierten Datenbanken* und kann als Eingabequelle oder Ausgabesenke für MapReduce Aufträge, Hive sowie Pig verwendet werden. Den Entwicklern nach eignet sich die Implementierung insbesondere für randomisierte Echtzeit Lese-/Schreibzugriffe auf großen Datenbeständen.

Hadoop Common

Das auch als *Core* bezeichnete Projekt stellt einige grundlegende Komponenten des Hadoop Systems sowie Schnittstellen für verteilte Dateisysteme zur Verfügung. Es werden zahlreiche Werkzeuge bereitgestellt, auf die alle Hadoop Unterprojekte zugreifen können.

2.2.2 MapReduce

Das *Hadoop MapReduce-Framework* ermöglicht es, Berechnungen automatisch auf mehreren Rechnern zu parallelisieren. Der Programmierer bekommt eine an funktionalen Programmiersprachen angelehnte Abstraktionsschicht zur Verfügung gestellt, die sich insbesondere durch ihre Einfachheit auszeichnet. Es sind lediglich zwei Funktionen zu implementieren, eine sogenannte *Map-* und eine *Reduce-Funktion*. Dieses Konzept hat den entscheidenden Vorteil, dass sich Entwickler auf das zu lösende Problem fokussieren können, ohne sich mit zahlreichen Details einer verteilten Anwendung beschäftigen zu müssen. Auch Hardwareausfälle werden von den Hadoop Entwicklern berücksichtigt. Fehlgeschlagene Aufgaben werden automatisch anderen Maschinen zugewiesen und erneut ausgeführt, ohne dass dieses Vorgehen explizit programmiert werden muss. Ebenso wenig muss in die Verteilung der Daten oder die Parallelisierung der Berechnungen eingegriffen werden. Zwar gibt es auch hierfür Möglichkeiten, diese sind jedoch für die klassischen Problemstellungen nicht erforderlich. Ein weiteres Grundprinzip dieses Programmiermodells betrifft die Art und Weise, wie mit Daten umgegangen wird. So werden, wie bei funktionalen Programmiersprachen üblich, Daten nicht manipuliert, sondern immer neu erzeugt. Des Weiteren spielt auch die Größe der zu verarbeiten Daten eine entscheidende Rolle. Das MapReduce Framework ist darauf ausgelegt, umfangreiche zusammenhänge Daten vollständig zu lesen und zu verarbeiten. Folglich müssen selbst diejenigen Anwendungen, die nur einen relativ kleinen Teil der Daten benötigen, ohne entsprechende Optimierungen die Eingabedatei vollständig einlesen.

Mapper & Reducer

Die Verarbeitungsschritte eines *MapReduce Auftrages* lassen sich, wie in Abbildung 5 dargestellt, in eine *Map-*, *Shuffle-* und *Reduce-Phase* einteilen. Bei der Ausführung eines MapReduce Auftrages wird zunächst die im *Hadoop Distributed Filesystem* (HDFS) abgespeicherte Eingabedatei in gleichgroße Blöcke unterteilt. In diesem Kontext werden solche Blöcke auch häufig als *Splits* be-

zeichnet. *Mapper*, die auf mehreren Rechnern verteilt instanziiert werden, bekommen nun diejenigen Splits als Eingabe, die auf den jeweiligen Rechnern lokal verfügbar sind. Hier kommt das zuvor erwähnte Data-Locality Konzept zum Tragen. Das heißt, Berechnungen werden zum Speicherort der Daten verschoben statt umgekehrt. Der Inhalt eines solchen Splits wird nun zeilenweise bearbeitet und als *Schlüssel-/Wertpaar* interpretiert. Im Folgenden wird ein solches Schlüssel-/Wertpaar auch als *Eintrag* bezeichnet. Die *Signatur* einer Map-Funktion ist dabei wie folgt festgelegt:

```
map(in_key, in_value) -> (out_key, intermediate_value) list
```

Da eine Map-Funktion aus den eingelesenen Einträgen eine neue Menge an Einträgen erzeugt, eignet sich diese Phase auch gut zur Datenvorbereitung. Werte, die fehlerhaft oder unvollständig sind, können verworfen werden. Auch ist es für eine Map-Funktion üblich, wenn der Informationsgehalt einer eingelesenen Zeile größer sein sollte als der Informationsbedarf einer bestimmten MapReduce Anwendung, nur die benötigen Teile einer Zeile zu extrahieren.

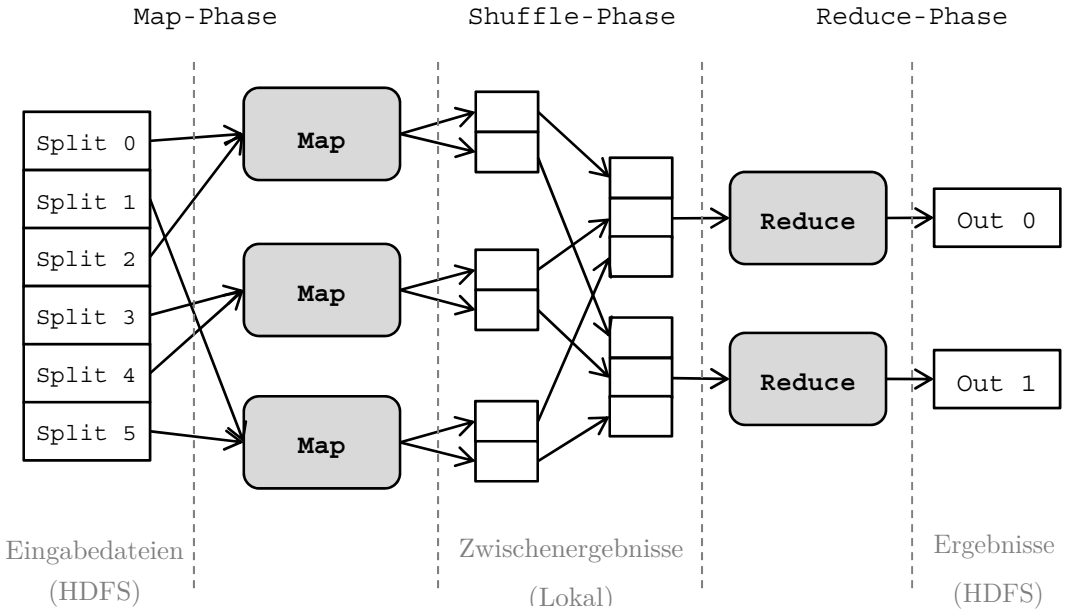


Abbildung 5: Verarbeitungsschritte einer MapReduce Anwendung

Weiterhin ist auch die Wahl des *Schlüssels* von großer Bedeutung. Während die Werte eines Eintrages als Informationsträger zu verstehen sind, haben die Schlüssel einen ausschlaggebenden Einfluss auf die weiterfolgende Bearbeitung der erzeugten Zwischenergebnisse. Diese werden im nächsten Schritt auf *Reducer* verteilt, wobei alle Einträge mit dem gleichen Schlüsselwert dem gleichen Reducer zugeführt werden. Hierfür werden, wie in Abbildung 5 schematisch dargestellt, in einer sogenannten Shuffle-Phase die Zwischenergebnisse zunächst

lokal vorsortiert und im nächsten Schritt, unter Verwendung einer geeigneten Hash-Funktion, den Reducern zugeordnet. Da die Reducer auf allen zur Verfügung stehenden Rechnern erzeugt werden können, müssen in dieser Phase häufig große Datenmengen über das Netzwerk transferiert werden. Nachdem die Verteilung der Daten abgeschlossen wurde, beginnt die eigentliche Reduce-Phase. Hierfür wird zu jedem Schlüsselwert eine *verkettete Liste* zugeordneter Werte erzeugt. Die Signatur einer Reduce-Funktion wird dabei wie folgt angenommen:

```
reduce(out_key, intermediate_value list) -> out_value list
```

Folglich werden in diesem Schritt alle Einträge, die in der Map-Phase den gleichen Schlüsselwert bekommen haben, zu einem einzigen Eintrag zusammengefasst. Die Reduce-Funktion wird für jeden solchen Eintrag gesondert aufgerufen, sodass sich ihre Aufgabe darauf einschränken lässt, die verketteten Werte eines einzigen Schlüssels verarbeiten zu können. Abschließend werden die von den Reducern erzeugten Ergebnisse im Hadoop Distributed Filesystem abgespeichert und können somit ausgegeben oder als Eingabe für weiterfolgende MapReduce Aufträge verwendet werden.

Beispiel

Im Folgenden verdeutlicht die beispielhafte Ausführung einer MapReduce Anwendung, wie ein konkretes Problem unter Verwendung einer Map- und Reduce-Funktion gelöst werden könnte. In Anlehnung an das Beispiel aus [White, 2009] soll aus einer Datei mit Temperaturmessungen, die über mehrere Jahre hinweg gesammelt worden sind, die maximal gemessene Temperatur für jedes Jahr bestimmt werden. Hierbei sollen lediglich diejenigen Messungen betrachtet werden, die in Deutschland durchgeführt wurden. Die zu verarbeitende Datei könnte beispielsweise folgendes Muster aufweisen:

```
2010/07/14,12:00,DE,79206,32Grad
2009/06/02,16:15,FR,38291,23Grad
1980/02/11,09:00,DE,78125,12Grad
...
```

Da eine beliebige Eingabedatei oftmals keine Schlüssel sondern nur Werte enthält, entspricht hier der Schlüssel dem Offset der entsprechenden Zeile. Eine Map-Funktion würde dann als Eingabe folgende Schlüssel-/Wertpaare bekommen:

```
(0, 2010/07/14,12:00,DE,79206,32Grad)
(80, 2009/06/02,16:15,FR,38291,26Grad)
(160, 1980/02/11,09:00,DE,78125,12Grad)
...
```

Entsprechend der Problemstellung werden aus jedem Eintrag nur das Jahr, das Land sowie die Temperatur extrahiert. Dabei wird jeder Eintrag, der nicht in Deutschland gemessenen wurde, verworfen. Für die Ausgabe der Map-Funktion wird in diesem Fall das Jahr als neuer Schlüssel und die gemessene Temperatur als neuer Wert verwendet. Die Zwischenergebnisse könnten dementsprechend wie folgt aussehen:

```
(1980, 12)
(1980, 21)
(2009, 23)
(2009, 29)
(2010, 32)
```

In der nachfolgenden Shuffle-Phase werden die Zwischenergebnisse unter Verwendung einer geeigneten Hash-Funktion an die entsprechenden Reducer-Instanzen übertragen. Anschließend werden alle Einträge mit gleichem Schlüsselwert, also alle Messungen, die im gleichen Jahr durchgeführt wurden, zu verketteten Listen zusammengefasst, sodass die Reduce-Funktionen folgende Schlüssel-/Wertpaare als Eingabe erhalten:

```
(1980, [12, 21])
(2009, [23, 29])
(2010, [32])
```

Da zu jedem Jahr eine Liste aller in Deutschland gemessenen Temperaturen erstellt wurde, müssen die Reduce-Funktionen nun für jedes Jahr über alle Temperatur-Einträge der Liste iterieren und dabei den maximalen Wert bestimmten, sodass folgende Ausgabe entsteht:

```
(1980, 21)
(2009, 29)
(2010, 32)
```

2.2.3 Hadoop Distributed Filesystem

Das *Hadoop Distributed Filesystem* (HDFS) wurde entwickelt, um Tera- bis Petabyte große Dateien verteilt in einem Rechencluster abzuspeichern. Da für die Rechencluster gewöhnliche Standard-Hardware verwendet werden sollte, die eine hohe Ausfallwahrscheinlichkeit aufweist, war eine gewisse Fehlertoleranz gegenüber Hardwareausfällen von entscheidender Bedeutung. Das HDFS ist daher so konzipiert, dass einzelne Rechner ausfallen können und sich der Anwender in der Regel nicht damit auseinandersetzen muss. Einem Datenverlust wird dabei durch eine mehrfache *Replikation* aller gespeicherten Daten vorgebeugt.

Weiterhin ist das HDFS darauf ausgelegt, zusammenhängende Datenbestände abzuspeichern. Deren Größe sollte dabei mindestens mehrere hundert Megabytes betragen und ist nach oben lediglich durch die Gesamtkapazität des Rechenclusters begrenzt. Auch sind Änderungen an beliebigen Stellen einer Datei im Nachhinein nicht mehr möglich. Das System ist demnach darauf ausgerichtet, Daten einmalig zu speichern und häufige zu lesen.

Die Speicherung der Daten erfolgt *blockbasiert*. Während jedoch bei typischen Dateisystemen die Blöcke nur mehrere hundert Kilobyte groß sein können, sind beim Hadoop Distributed Filesystem Blockgrößen von 64 bis 128 MB üblich. Dies wirkt sich zum einen positiv auf die Übertragungsrate der Festplatte aus, da deutlich weniger zufällige Zugriffe[12] notwendig werden. Zum anderen reduziert sich der Aufwand für die Verwaltung der Blöcke und das gesamte Speichersystem wird einfacher. Ein weiterer, wesentlicher Vorteil begründet sich mit der im vorherigen Kapitel beschriebenen Unterteilung der Eingabedatei in gleichgroße Splits für die Map-Funktionen. Im Regelfall entsprechen diese Splits genau den hier vorgestellten Blöcken.

Ein *HDFS Cluster* besteht aus einem *Namenode* und zahlreichen *Datanodes*. Der Namenode verwaltet die Verzeichnisstrukturen sowie die Metainformationen aller im HDFS abgelegten Dateien[13]. Die tatsächlichen Daten-Blöcke einer Datei werden auf die Datanodes verteilt. Die Hauptaufgabe des Namenodes besteht nun darin, die Speicherorte der Blöcke einer angefragten Datei zu liefern. Um diese Anfrage möglichst schnell beantworten zu können, müssen alle Informationen stets im Arbeitsspeicher des Namenodes gehalten werden. Das

[12] Der Zugriff auf einen neuen Block erfordert, dass der Schreib-/Lesekopf einer Festplatte neu ausgerichtet werden muss. Dieser Vorgang benötigt ca. 5 – 10 ms.

[13] Ein sekundärer Namenode verfolgt alle Änderungen am primären Namenode und sorgt für eine Ausfallsicherheit der Metainformationen.

hat zur Folge, dass häufig lediglich wenige Millionen Dateien gleichzeitig verwaltet werden können.[14]

Abschließend sei noch angemerkt, dass in Verbindung mit dem MapReduce Framework das Hadoop Distributed Filesystem zwar in vielen Fällen das zu favorisierende Dateisystem darstellt, Hadoop jedoch die Anbindung zahlreicherer anderer Speichersysteme wie beispielsweise *Amazons S3, FTP-* oder *WebDAV-Server* ermöglicht.

2.3 Berkeley Database

Bei der *Berkeley Database (Berkeley DB)* handelt es sich um eine von Oracle frei angebotene *Datenbank-Bibliothek*, die direkt in Anwendungen eingebettet werden kann. Sie wird von namhaften Unternehmen wie beispielsweise Yahoo, Amazon, Google und Cisco Systems in Mobiltelefonen, Routern sowie diversen Programmen zur Datenspeicherung verwendet. Bis vor kurzem war ein Zugriff auf die Daten nur unter Verwendung diverser Programmierschnittstellen möglich wie beispielsweise C, C++, Java, Perl und Python. Neuerdings lässt sie sich auch mit *SQL-Statements* bedienen[15]. Eingebettet in eine Anwendung ist kein separater Server oder zusätzlicher administrativer Aufwand notwendig.

Im Gegensatz zu den meisten anderen Datenbankmanagementsystemen beruht die Berkeley DB nicht auf einem relationalen Datenmodell. Anstatt mehrspaltige Tabellen zu verwalten, werden lediglich Schlüssel-/Wertpaare abgespeichert. Dementsprechend fällt auch die Anzahl der unterstützten Operationen relativ gering aus. Neben dem *Einfügen* und dem *Entfernen* eines Eintrages sind nur das *Suchen* nach einem bestimmten Schlüssel sowie die *Aktualisierung* eines gefundenen Eintrages möglich. Eine Anwendung kann dabei zwischen unterschiedlichen Datenstrukturen wählen, die persistent auf der Festplatte oder flüchtig im Arbeitsspeicher abgelegt werden. Unterstützt werden, neben *B-Bäumen*, außerdem noch *Hashtabellen* sowie *persistente Warteschlangen*.

[14] Je Datei, Block und Verzeichnis werden ca. 150 Bytes benötigt. Bei Hadoop-Clustern mit mehreren tausend Rechnern ist es üblich Namenodes mit 100 GB Arbeitsspeicher einzusetzen.

[15] http://www.oracle.com/technology/products/berkeley-db/sql.html

Weiterhin verfügt die Berkeley DB über mehrere, aus Datenbanksystemen bekannte, Funktionalitäten. So werden *Transaktionsmechanismen* mit *ACID Eigenschaften*[16] unterstützt. Auch können mehrere Benutzer konkurrierend auf die Datenbank zugreifen. Die Berkeley DB verfügt über entsprechende *Sperrmechanismen*, die transparent für den Anwender durchgeführt werden. Zwei weitere Eigenschaften betreffen die Verwendung des Arbeitsspeichers. Da die Berkeley DB aus einer Anwendung heraus initialisiert wird, nutzen Anwendung und Datenbank einen gemeinsamen Speicherraum. Weiterhin steht der Datenbank ein Cache zur Verfügung, der in seiner Größe automatisch verwaltet oder explizit gesetzt werden kann.

Für eine weitergehende Vertiefung ist die von Oracle bereitgestellte Dokumentation [Oracle, 2010] zu empfehlen. Informationen zu der im Rahmen dieser Arbeit gewählten Konfiguration sowie dem Umgang mit der Berkeley DB werden im Zuge der Implementierung des *Dictionary Encodings* in Abschnitt 4.5 näher erläutert.

[16] Im Deutschen auch AKID genannt (Atomarität, Konsistenzerhaltung, Isoliertheit, Dauerhaftigkeit)

3 Pfadanfragesprache RDFPath

Da bei der Analyse von RDF-Graphen insbesondere *Pfade* betrachtet werden sollten, wurde im Rahmen dieser Arbeit die *Pfadanfragesprache RDFPath* entworfen. In diesem Kapitel werden zunächst einige allgemeine Anforderungen definiert, die bei der Konzeption der Sprache berücksichtigt werden sollten. Daran anschließend werden zwei Vorschläge für eine solche Sprache vorgestellt und eine Abgrenzung zu verwandten Anfragesprachen vorgenommen. Der Hauptteil setzt sich mit einer Einführung in die Syntax und Semantik von RDFPath auseinander und vervollständigt das Kapitel mit einigen abschließenden Beispielen.

Die Anfragesprache stellt ein zentrales Werkzeug für die Analyse von RDF-Graphen dar und ist infolgedessen auch ein wichtiger Aspekt der Implementierung. Als Bindeglied zwischen der Problemstellung und der tatsächlichen Realisierung, muss die Sprache dabei zwei Seiten gerecht werden.

Da die Analyse von RDF-Graphen über die Spezifikation von Pfaden erfolgen sollte, war zunächst die Erfüllung aller wesentlichen Kriterien einer Pfadanfragesprache erforderlich. Dementsprechend sollte es möglich sein, eine *Menge an Pfaden*, bestehend aus Kanten und Knoten des RDF-Graphen, deklarativ zu spezifizieren. Dabei sollten sowohl die Reihenfolge, in der die Kanten verfolgt werden, als auch die Anzahl der Kanten, denen nachgegangen wird, frei wählbar sein. Ebenso sollte es auch möglich sein, Eigenschaften von Knoten zu prüfen, die im Kontext von Pfaden als *Filter* verstanden werden können. Des Weiteren sollte auch die *Ausgabeart* variiert werden können. So sollten neben allen erreichbaren Knoten auch die dazugehörigen Pfade oder nur die jeweilige Länge ausgegeben werden können.

Auf der anderen Seite galt es, die Anbindung an die Implementierung auf MapReduce Ebene zu betrachten. Die unterstützten Anfragekonstrukte sollten in eine *Sequenz von MapReduce Aufträgen* überführt werden können, welche die Problemstellung möglichst effizient lösen. Um jedoch die Problemstellungen möglichst effizient lösen zu können, musste die Art und Weise, wie Anfragen ausgedrückt werden, der typischen Vorgehensweise entgegenkommen, wie Probleme auf MapReduce Ebene verteilt gelöst werden. Andernfalls wären zusätzliche Optimierungen in Erwägung zu ziehen, wie es beispielsweise bei der Überführung von *SPARQL* nach *Pig Latin* der Fall ist [Schätzle, 2010].

Entsprechend dieser Anforderungen entstanden im Rahmen dieser Arbeit zwei Vorschläge für eine Pfadanfragesprache, die im Folgenden näher betrachtet werden. Da die Entscheidung zu Gunsten der zweiten Anfragesprache ausgefallen ist, wird der erste Vorschlag lediglich in seinen Grundzügen kurz besprochen.

3.1 Pattern Matching

Die Idee hinter diesem Ansatz beruht auf der wiederholten Anwendung eines *Musters* auf eine *Menge an Triples*, wobei die Pfade mit jeder Anwendung dieses Musters um einen Schritt wachsen. Eine Menge an Triples wird dabei über die Deklaration von *festen* und *variablen Ressourcen* festgelegt. Hierfür können bei einem Triple der Form `(Subjekt, Prädikat, Objekt)` beispielsweise das Subjekt und das Prädikat fest vorgegeben und das Objekt variabel gelassen werden, wobei variable Ressourcen durch einen *Stern* dargestellt werden. Zu einer vollständigen Anfrage gehören, wie in Abbildung 6 zu sehen ist, drei Komponenten. Die erste Komponente stellt das Triple dar, welches die festen und variablen Ressourcen vorgibt, im Folgenden auch *Pattern* genannt. Der zweite Bestandteil einer Anfrage gibt an, wie oft das Pattern angewendet werden soll und als Letztes wird die gewünschte *Ausgabeart* spezifiziert. Umgangssprachlich ausgedrückt werden mit dieser beispielhaften Anfrage alle Knoten bestimmt, die ausgehend vom Startknoten `„Sebastian“` über drei Kanten mit dem Prädikat `„kennt“` erreicht werden können.

```
PATTERN (Sebastian, kennt, *)
CALLS   3
OUTPUT  Node
```

Abbildung 6: Anfrage in einer Pattern Matching Sprache

Bei der ersten Anwendung des Patterns dient der vollständige RDF-Graph, der durch eine Menge von RDF-Triples beschrieben wird, als Eingabe. Alle Triples, die dem Pattern entsprechen, werden in die *Menge der Zwischenergebnisse* aufgenommen. Demzufolge wären das bei der Anfrage aus Abbildung 6 alle Triple mit dem Subjekt `„Sebastian“` und dem Prädikat `„kennt“`. Für die nächste Anwendung dieses Patterns wird das feste Subjekt `„Sebastian“` durch genau die Menge an Objekten ersetzt, die in der vorrangegangenen Anwendung des Patterns als Zwischenergebnisse aufgenommen wurden. Diesem Prinzip folgend wird das Pattern solange ausgewertet, bis die gewünschte Anzahl an Wiederholungen erreicht ist.

Die bisherige Beschreibung ist für komplexere Problemstellungen ungeeignet und sollte auch lediglich den konzeptuellen Ansatz verdeutlichen. In der nachfolgenden Abbildung 7 ist ein etwas umfangreicheres Beispiel mit einigen Erweiterungen zu sehen.

```
PATTERN ((Sebastian, kennt, *, 2), wohnort, * , 1)
OUTPUT  Path
```

Abbildung 7: Anfrage in einer Pattern Matching Sprache (2)

Zum einen können Patterns ineinander verschachtelt werden, wobei die Ergebnisse des inneren Patterns für die Auswertung des äußeren benötigt werden. Außerdem hat die Verwendung mehrerer Patterns zur Folge, dass die Anzahl an Wiederholungen, die an ein bestimmtes (Teil-)Pattern gebunden sind, beim jeweiligen Pattern angegeben werden müssen. Ein Pattern hat folglich die Struktur:

`(Subjekt, Prädikat, Objekt, Anzahl Wiederholungen)`

Im aktuellen Beispiel wird nach allen Knoten gefragt, die ausgehend vom Startknoten „`Sebastian`“ über die Kante „`kennt`“ in genau zwei Schritten erreicht werden können. Ausgehend von den hierdurch spezifizierten Knoten soll durch die Anwendung des nächsten Patterns der Kante „`wohnort`“ ein Mal gefolgt werden. Die beispielhafte Ausgabe einer solchen Anfrage könnte wie folgt aussehen:

```
Sebastian (kennt) Franz (kennt) Sandra (wohnort) Freiburg
  Sebastian (kennt) Sandra (kennt) Simon (wohnort) Basel
                          ...
```

Auch wenn sich mit diesen Erweiterungen bereits einige interessante Fragestellungen beantworten lassen, fehlt es beispielsweise an Möglichkeiten, Knoten auf bestimmte Eigenschaften zu prüfen oder *Unteranfragen* auszudrücken. Entsprechende Erweiterungen ließen sich zwar modellieren, hätten jedoch eine deutlich komplexere Syntax zur Folge, die bereits in dieser Fassung den Anforderungen bezüglich Intuition und Lesbarkeit nicht gerecht wurde. Der nächste Vorschlag für eine Pfadanfragesprache setzt genau an diesen beiden Kritikpunkten an. Die Anfragen sollten vor allem intuitiver und bezüglich ihrer Ausdrucksmöglichkeiten einfacher zu erweitern sein.

3.2 RDFPath

RDFPath ist eine Pfadanfragesprache, die – entsprechend den zuvor erläuterten Anforderungen – insbesondere im Hinblick auf die spätere Überführung in eine *Sequenz von MapReduce Aufträgen* konzipiert wurde. Pfade werden, in Anlehnung an *XPath*, aus mehreren *Lokationsschritten* zusammengesetzt. Ausgehend von einem Knoten oder einer Knotenmenge betrachtet jeder Lokationsschritt die Möglichkeiten, über eine vorher festgeschriebene Kante weitere Knoten zu erreichen. Folglich wird für die Auswertung des *i+1'ten* Lokationsschrittes als Eingabe zum einen die Ausgabe des *i'ten* Lokationsschrittes und zum anderen der ursprüngliche RDF-Graph benötigt. Weitergehende Informationen, welche die konkrete Auswertung eines Lokationsschrittes sowie die Umsetzung in MapReduce Aufträge betreffen, werden in Abschnitt 4.3 besprochen.

Je nach Verwendungszweck gibt es unterschiedliche Vorstellungen von einer Anfragesprache für RDF-Graphen. Projekte wie Hive zeigen dabei, dass sich die aus relationalen Datenbanken bewährte *SQL-Syntax* auch im Bereich von MapReduce einer sehr großen Beliebtheit erfreut und folglich auch hier eine immer größer werdende Bedeutung einnimmt. Um jedoch die Besonderheiten des RDF-Modells zu berücksichtigen, das weniger einem klassischen relationalen Datenmodell als vielmehr einem (verbundenen) gerichteten Graphen entspricht, sind neuere Ansätze wie beispielsweise *SPARQL* oder spezielle Pfadanfragesprachen zu betrachten.

Bei *XPath* handelt es sich um eine Sprache, die nach [Lausen, 2005] hauptsächlich zur Adressierung von Teilen eines *erweiterten XML-Teilbaumes* gedacht ist. Im Gegensatz zu RDFPath wird jedoch eine für *XML-Dokumente* typische *Dokumentenstruktur* vorausgesetzt, die beispielweise einen eindeutigen *Wurzelknoten* vorsieht. Dagegen können in RDFPath die *Referenzknoten* bzw. *Startknoten* frei gewählt werden. Weiterhin ist für die Identifikation benachbarter Knoten in RDFPath keine Unterscheidung zwischen den *Kantenarten* erforderlich, wohingegen XPath zu diesem Zwecke ein hierarchisches Konzept mit unterschiedlichen *Achsenbewegungen* anbietet. Ihre Gemeinsamkeiten haben beide Sprachen im *objektorientierten Ansatz* der Lokationsschritte, mit denen die Pfade spezifiziert werden.

SPARQL ist eine vom World Wide Web Consortium (W3C) veröffentlichte Anfragesprache, um RDF-Graphen zu analysieren. Hierfür stehen dem Anwender beispielsweise *optionale Graph Patterns*, *Filter* und *Joins* als Werkzeuge zur Verfügung. Nach [Pérez, et al., 2008] verfügt SPARQL jedoch nur über eingeschränkte *Navigationsmöglichkeiten*, sodass sich längere Pfade nur um-

ständlich ausdrücken lassen. RDFPath hingegen ist genau für diese Szenarien konzipiert worden und ermöglicht es Pfadanfragen über RDF-Graphen auf eine einfache Art zu konstruiert. Auch die *Suche nach kürzesten Pfaden*, die in SPARQL nicht ausgedrückt werden kann, wurde mit RDFPath abgedeckt. Abgesehen von der fehlenden Unterstützung für Pfadanfragen stellt SPARQL allerdings ein sehr universelles Analysewerkzeug für RDF dar. Folglich kann RDFPath auch als eine Ergänzung zu SPARQL verstanden werden, die zusätzliche Navigationsmöglichkeiten bereitstellt. Entsprechende Vorschläge wurden beispielsweise in [Alkhateeb, et al., 2009] mit *PathRDF* und in [Pérez, et al., 2008] mit *nSPARQL* vorgestellt. Im Gegensatz zu diesen Vorschlägen lag der Schwerpunkt bei RDFPath jedoch nicht auf der Ergänzung von SPARQL, sondern auf den Besonderheiten des MapReduce Frameworks. Dies zeigt sich insbesondere bei den einzelnen Komponenten dieser Sprache und den daraus resultierenden *Ausdrucksmöglichkeiten*, die im nächsten Abschnitt besprochen werden.

3.3 Syntax

Der Syntax von RDFPath liegt das zuvor eingeführte Konzept von Lokationsschritten zugrunde. Eine Anfrage setzt sich dabei aus einem Startknoten gefolgt von einer *Sequenz an Lokationsschritten* zusammen. Im Folgenden werden die Bestandteile von RDFPath nacheinander eingeführt und abschließend eine Syntax in der *erweiterten Backus-Naur-Form (EBNF)*[17] vorgestellt.

Startknoten

Der *Startknoten* stellt den Referenzpunkt für die Auswertung einer Pfadanfrage dar und wird folglich auch an erster Stelle spezifiziert.

```
Peter :: kennt .              (1)
  *   :: wohnort .            (2)
```

Abbildung 8: Startknoten in RDFPath

Wie in Abbildung 8 (1) zu sehen ist, wird er durch die Zeichenfolge „`::`“ von dem weiter rechts aufgeführten Lokationsschritt getrennt. In dieser Anfrage wird nach allen Pfaden gefragt, die ausgehend vom Knoten „`Peter`“ der Kante „`kennt`“ folgen. Eine beispielhafte Ausgabe könnte sich nun wie folgt aus mehreren Pfaden mit der Länge eins zusammensetzen:

[17] ISO/IEC 14977 : 1996(E) – ISO Standard zu EBNF

```
Peter (kennt) Simon
Peter (kennt) Sarah
```

Statt einen Knoten eindeutig mit seinem Bezeichner zu identifizieren, ist es unter Verwendung des „*"-*Operators* auch möglich beliebige Startknoten vorzusehen. Dies wird in Anfrage (2) verdeutlicht. Ein kurzer Einblick auf eine mögliche Implementierung verdeutlicht die Auswirkung dieses Operators. Der Startknoten fungiert hierbei wie eine Art *Filter*. Dabei werden alle Triples verworfen, deren Subjekte einen anderen Bezeichner aufweisen als über den Startknoten spezifiziert wurde. Der „*"-Operator setzt nun diese Filterung außer Kraft, sodass alle betrachteten Triples als Pfade in die Ergebnismenge aufgenommen werden. Dabei werden jedoch nur diejenigen Triples betrachtet, welche die im ersten Lokationsschritt angegebene Kante (Prädikat) enthalten. Das hier zugrunde liegende Datenkonzept wird in Kapitel 4.1 detailliert eingeführt. Eine beispielhafte Ausgabe für die Anfrage (2) könnte folgendermaßen aussehen:

```
Peter (wohnort) Karlsruhe
Michael (wohnort) Freiburg
```

Lokationsschritte

Die *Lokationsschritte* bilden die grundlegende *Navigationskomponente* in RDFPath. Sie spezifizieren sowohl die Reihenfolge, in der die Kanten verfolgt werden, als auch die Anzahl der Kanten, denen nachgegangen wird. Sie werden, wie in Abbildung 9 dargestellt, mit dem Zeichen „>" voneinander getrennt, wobei der letzte Lokationsschritt mit dem *Terminalsymbol* „." die Anfrage abschließt.

```
Peter :: kennt > wohnort > plz .
```

Abbildung 9: Trennung von Lokationsschritten

In allen Bestandteilen einer Pfadanfrage, zu denen neben der Angabe von Knoten und Kanten auch die als nächstes eingeführten Ergebnis- sowie Filterausdrücke zählen, findet keine Unterscheidung zwischen Groß- und Kleinbuchstaben statt. Auch die Anzahl der Leerzeichen spielt keine Rolle. Anfragen können folglich je nach Belieben auch als zusammenhängende Zeichenketten formuliert werden.

Ergebnisausgaben

In den bisherigen Beispielen setzten sich die *Ergebnismengen* immer aus vollständigen *Pfaden* zusammen, die ausgehend vom Startknoten alle besuchten

Kanten und Knoten repräsentieren. Das ist die *Standardausgabe*, falls keine zusätzlichen Angaben vorgenommen werden. In Tabelle 1 sind alle implementierten *Ausgabearten* mit einer kurzen Beschreibung sowie einem Beispiel aufgeführt.

Die Verwendung der *Aggregationsmethoden* `AVG()`, `SUM()`, `MAX()`, `MIN()`, `COUNT()` ist an einige Voraussetzungen gebunden. So können diese Methoden nur dann verwendet werden, wenn der letzte Knoten in allen betrachteten Pfaden einen Zahlenwert, wie beispielsweise das Alter, repräsentiert. Für Zeichenketten würde die Ergebnismenge leer bleiben.

Kodierung	Beschreibung	Beispielausgaben[18]

`PATH()`	Liefert vollständigen Pfad (Standardausgabe)	`5 (kennt) 7`
		`5 (kennt) 2`
		`5 (kennt) 3 (kennt) 7`

`NODES()`	Liefert den letzten Knoten eines Pfades	`7`
		`2`
		`7`

`COUNT()`	Liefert die Anzahl an Ergebnis-Pfaden Es werden keine Pfade ausgegeben	`3`
`LIMIT(Int)`	Begrenzt die Ausgabe an ausgegebenen Pfaden	**`LIMIT(1):`** `5 (kennt) 7`

`DISTANCE(Node)`	Liefert die Entfernung und den vollständigen Pfad zum Knoten „Node“	**`DISTANCE(7):`**
		`1: 5 (kennt) 7`
		`2: 5 (k) 3 (k) 7`

`AVG()`	Liefert den Durchschnitt der Werte, den die letzten Knoten repräsentieren	`5`
`SUM()`	Liefert die Summe der Werte, den die letzten Knoten repräsentieren	`16`
`MAX()`	Liefert das Maximum der Werte, den die letzten Knoten repräsentieren	`7`
`MIN()`	Liefert das Minimum der Werte, den die letzten Knoten repräsentieren,	`2`
`ALL()`	`COUNT() +AVG() + SUM() + Max() + MIN()`	`count:3, avg:5, sum:16, max:7, min:2`

Tabelle 1: Ergebnisarten von RDFPath

[18] Vereinfachte Darstellung

In Abbildung 10 ist die syntaktische Verwendung von unterschiedlichen Ergebnisausgaben dargestellt. Die Ergebnisart wird nach dem Terminalsymbol wie ein *Funktionsaufruf* spezifiziert.

```
Peter :: kennt > hatTelefonNr .NODES()
Peter :: kennt > hatAlter .AVG()
Peter :: kennt > kennt .LIMIT(1000)
```

Abbildung 10: Ergebnisse in RDFPath

Filter

Innerhalb eines Lokationsschrittes können neben der Angabe der zu betrachtenden Kante noch *Bedingungen* an die Knotenbezeichner ausgedrückt werden. Diese Bedingungen können analog zum Startknoten auch als Filter verstanden werden. Tabelle 2 zeigt eine Übersicht der in RDFPath implementierten *Filterausdrücke*.

Kodierung	Beschreibung	Beispielausgaben[19]
`EQUALS(String)`	Knoten muss die gleiche Zeichenkette aufweisen	**`EQUALS(Berlin):`** `Berlin`
`PREFIX(String)`	Knoten muss mit der entsprechenden Zeichenkette beginnen	**`PREFIX(Be):`** `Berlin, Berkeley`
`SUFFIX(String)`	Knoten muss mit der entsprechenden Zeichenkette enden	**`SUFFIX(html):`** `s1.html, s2.html`
`MIN(Int)`	Knoten muss Zahlenwert darstellen und mindestens den angegebenen Wert aufweisen	**`MIN(1997):`** `1997, 1998, 2010`
`MAX(Int)`	Knoten muss Zahlenwert darstellen und maximal den angegebenen Wert aufweisen	**`MAX(2010):`** `1997, 2009, 2010`

Tabelle 2: Filter in RDFPath

Wie bei der Verwendung von unterschiedlichen Ergebnisarten, resultiert auch hier der falsche Einsatz eines Filterausdrucks nicht in einer Fehlermeldung, sondern in der Filterung des aktuell betrachteten Eintrages. Dieses Verhalten ist insbesondere bei unvollständigen oder fehlerhaften Datensätzen vorteilhaft. Entsprechende Einträge, bei denen die Anwendung der gewählten Filterung

[19] Beispiele, die den fett markierten Filterausdruck erfüllen würden.

fehlschlägt, werden verworfen und haben somit keine Auswirkung auf die weitere Auswertung der Anfrage.

Wie in Abbildung 11 (1) zu sehen ist, werden Filter nach der Spezifikation einer Kante in eckigen Klammern ausgedrückt. Bei mehr als einem Filter werden die entsprechenden Filterausdrücke nacheinander (2) angegeben.

```
* :: geburtsjahr [max(1991)] > name.                    (1)
Peter :: kennt > wohnort [prefix(Be)] [suffix(burg)].   (2)
```

Abbildung 11: Filterausdrücke in RDFPath

Unteranfragen

Während die zuvor besprochenen Filter lediglich auf Knoten operieren können, die im aktuellen Lokationsschritt erreicht wurden, erlauben *Unteranfragen* die Betrachtung aller über eine beliebige Kante erreichbaren Knoten. In ihrer Anwendung wirken sie wie Filter, die Knotenbezeichner auf bestimmte Eigenschaften prüfen. Hierfür stehen auch alle Ausdrücke aus der Tabelle 2 zur Verfügung, wobei die zusätzliche Angabe der zu betrachtenden Kante erforderlich ist. Wenn eine Bedingung nicht erfüllt werden kann, wird der entsprechende Pfad verworfen. Dies ist auch der Fall, falls ein Knoten über keine ausgehende Kante verfügt, die mit der Unteranfrage betrachtet werden soll.

Für die Auswertung einer Unteranfrage werden die im aktuellen Lokationsschritt betrachteten Knoten als *Referenzknoten* verwendet. Für die Unteranfrage in Abbildung 12 (1) wären das beispielsweise diejenigen Knoten, die über die Kante „wohnort“ erreicht wurden. Von diesen Knoten ausgehend wird die Kante „plz“ verfolgt und die erreichten Knoten entsprechend des angegeben Filterausdrucks geprüft. Umgangssprachlich würde das bedeuten, dass bei dieser beispielhaften Anfrage nur diejenigen Wohnorte zulässig sind, deren Postleitzahl mit „79“ anfängt. Wohnorte ohne Postleitzahl würden verworfen werden. Für den nachfolgenden Lokationsschritt, im aktuellen Beispiel (1) wäre das die Betrachtung der Kante „einwohnerzahl“, werden als Referenzknoten weiterhin die über die Kante „wohnort“ erreichten Knoten betrachtet.

Wie in Abbildung 12 (2) zu sehen ist, können auch Unteranfragen, analog zu den Filtern, mehrmals nacheinander vorkommen.

```
Peter :: kennt > wohnort [plz = prefix(79)] > einwohnerzahl.   (1)
Peter :: wohnort [plz = prefix(79)] [einwohnerzahl = min(30].  (2)
```

Abbildung 12: Unteranfragen in RDFPath

Abkürzungen

Bei der Konstruktion längerer Pfadanfragen ist die Verwendung einer *abkürzenden Schreibweise* möglich. Dies funktioniert allerdings nur dann, wenn identische Lokationsschritte nacheinander ausgeführt werden sollen. In solchen Fällen können die Lokationsschritte, wie in Abbildung 13 (1) zu sehen ist, durch einen einzigen Ausdruck (2) ersetzt werden. Die Anzahl an Wiederholungen wird dabei innerhalb runder Klammern am Ende des spezifizierten Lokationsschrittes angegeben. Es ist wichtig anzumerken, dass hier der vollständige Lokationsschritt mit Filter und einer beliebigen Anzahl an Unteranfragen mehrfach ausgeführt wird.

```
Peter :: kennt > kennt > kennt.          (1)
Peter :: kennt (3).                      (2)
```

Abbildung 13: Abkürzende Schreibweise in RDFPath

Kürzeste Pfade

Die Bestimmung des *kürzesten Pfades* zwischen zwei Knoten ist ein klassisches Problem aus der Graphen Theorie. In RDFPath können entsprechende Anfragen unkompliziert ausgedrückt werden. Hierfür wird auf das zuvor eingeführte Konzept einer abkürzenden Schreibweise zurückgegriffen, welches für die Berechnung von kürzesten Pfaden um einen *„*"-Operator* erweitert wird. Der Operator wird immer mit einem Zahlenwert aufgeführt. Dieser Wert ist als *obere Grenze* für die *maximale Suchtiefe* zu verstehen. Folglich werden in Abbildung 14 (1) alle Knoten betrachtet, die ausgehend von Peter über maximal drei Kanten „kennt" erreichbar sind.

```
Peter :: kennt (*3).                                (1)
Peter :: kennt [wohnort = equals(Berlin)] (*2).     (2)
Peter :: kennt (*8) .DISTANCE(Simon)                (3)
```

Abbildung 14: Kürzeste Pfade in RDFPath

Das heißt die potentielle Ergebnismenge setzt sich in diesem Beispiel aus Pfaden der Länge eins bis drei zusammen. Dabei wird bei Knoten, die über unterschiedliche Pfade erreichbar sind, immer der kürzere Pfad weiter betrachtet und die längeren Pfade werden verworfen. Eine wichtige Voraussetzung für diese Berechnung sind dabei feste Startknotenbezeichner. Demnach ist hier der „*"-Operator als Startknoten nicht zulässig. Kürzeste Pfade können flexibel in längere Pfadanfragen eingebettet werden, denn jeder beliebige Lokationsschritt darf kürzeste Pfade berechnen. Hierbei ist es nicht von Bedeutung, ob dieses

spezielle Werkzeug am Anfang, in der Mitte oder am Ende einer Sequenz an Lokationsschritten benötigt wird. Ebenso ist auch die Berechnung von mehreren kürzesten Pfaden, die unterschiedliche Kanten verfolgen, zulässig. Weiterhin ist, wie in Abbildung 14 (2) dargestellt, eine Kombination mit Filtern oder Unteranfragen möglich. Dabei haben Unteranfragen, die notwendigerweise eine weitere Kante betrachten müssen, keine Auswirkung auf die Länge des Pfades.

Ferner ist auch die Kombination mit der Ergebnisart „`DISTANCE()`" interessant. So würde die Anfrage (3) aus dem aktuellen Beispiel umgangssprachlich unter allen Personen, die Peter über maximal acht Kanten kennt, den kürzesten Pfad zu Simon bestimmten. Im Gegensatz zum vorherigen Beispiel, in dem die kürzesten Pfade zu allen erreichbaren Knoten ausgegeben wurden, wird hier nur ein kürzester Pfad, der bei Simon endet, betrachtet.

Zyklen

Bei der Auswertung von Pfadanfragen kann es durchaus auftreten, dass ein Knoten entlang eines Pfades mehrfach besucht wird. Diese auch als *Zyklus* bekannte Folge an Knoten und Kanten hat in der Regel unerwünschte Auswirkungen auf die Ergebnismenge. Folglich bedarf es Mechanismen, die während der Auswertung entsprechende *Prüfungen* durchführen. Hierbei hat es sich als sinnvoll erwiesen, unterschiedliche *Prüfungsstrategien* zu betrachten. Die implementierten Regeln sind in Tabelle 3 aufgelistet.

Kodierung	Beschreibung	Beispiel [20]
`ALL`	Zyklen sind erlaubt	`3 (k) 4 (k) 3`
`NONE`	Zyklen sind nicht erlaubt	~~`3 (k) 4 (k) 3`~~
`EDGES`	Zyklen über gleiche Kanten sind nicht erlaubt Zyklen über unterschiedliche Kanten sind erlaubt	~~`3 (k) 4 (k) 3`~~ `3` (k_1) `4` (k_2) `3`

Tabelle 3: Zyklenerkennung in RDFPath

Da jedoch die Einbettung einer solchen Funktionalität in die Pfadanfrage weitreichende Änderungen der Syntax zur Folge hätte, wird der gewünschte Umgang mit Zyklen gesondert von der eigentlichen Anfrage spezifiziert.

[20] `(k)`: Abkürzende Schreibweise für eine Kante
$k_1 \neq k_2$

Grammatik

Die *erweiterte Backus-Naur-Form (EBNF)*[21] ist eine formale Metasprache, um *kontextfreie Grammatiken* darzustellen. Sie wird in der nachfolgenden Tabelle 4 verwendet, um die Syntax von RDFPath formal zu definieren. Da bei einer Grammatik üblicherweise englische Bezeichner verwendet werden, wurde hier auf die Verwendung der deutschen Begriffe verzichtet und stattdessen die englischen Übersetzungen herangezogen.

```
Query          = startnode '::' locationstep  '.' [ resultfunction ] .
locationstep   = edge [ condition ] [ repeat | shortestpath ]
                 [ '>' locationstep ] .
condition      = ( filter | subquery ) [ condition ].
filter         = '[' filterfunction ']' .
filterfunction = 'EQUALS(' textvalue ')' | 'PREFIX(' textvalue ')'
                 | 'SUFFIX(' textvalue ')'
                 | 'MIN(' intvalue ')' | 'MAX(' intvalue ')' .
subquery       = '[' edge '=' filterfunction ']' .
repeat         = '(' intvalue ')' .
shortestpath   = '(*' intvalue ')' .
resultfunction = 'PATH()' | 'NODES()' | 'COUNT()' | 'AVG()'
                 | 'SUM()' | 'MAX()' | 'MIN()' | 'ALL()'
                 | 'LIMIT(' intvalue ')' | 'DISTANCE(' textvalue ')' .
startnode      = '*' | textvalue .
edge           = textvalue .
textvalue      = { legalChars } .
intvalue       = { allDigits } .
legalChars     = allChars - reservedChars .
reservedChars  = '::' | '>' | '(' | ')' | '[' | ']' | ' ' .
allDigits      = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
                 | '9' .
allChars       = ? all visible characters ? .
(* filterfunction and resultfunction are case intensitive *)
```

Tabelle 4: Syntax von RDFPath in EBNF

[21] ISO/IEC 14977 : 1996(E) – ISO Standard zu EBNF

3.4 Semantik

In diesem Abschnitt wird eine Semantik für RDFPath vorgestellt, die in Anlehnung an zwei in [Wadler, 1999] vorgeschlagene Semantiken für XPath konzipiert wurde. Für eine kurze Einführung in die hier zugrunde liegende *denotationelle Semantik* ist [Wadler, 2000] zu empfehlen. Als Vereinfachung wird im Folgenden auf die Darstellung von kürzesten Pfaden, dem „`*`"-Operator als Startknoten sowie unterschiedlichen Ergebnisvarianten, die in der Syntax auch mit „`resultfunctions`" betitelt wurden, verzichtet.

Ein $\boldsymbol{Graph\ G = (V, E)}$ besteht aus einer Menge V an Knoten und einer Menge E an Kanten. Der Knoten $v_2 \in V$ gilt als Nachbar des Knotens $v_1 \in V$ wenn es in G eine gerichtete Kante $e = (v_1, v_2),\ e \in E$ von v_1 nach v_2 gibt.

Jedem Knoten $v \in V$ sei ein Bezeichner $label(v)$ zugeordnet und jeder Kante $e \in E$ sei ein Bezeichner $\boldsymbol{label(e)}$ zugeordnet und. Im Kontext von RDF kann $label(e)$ auch als Prädikat und $label(v)$ als Objekt oder Subjekt verstanden werden.

Sei im Folgenden $\boldsymbol{v(value_2)} = \{\, v \in V \mid label(\, v\,) = value_2 \,\}$ die Menge an Knoten mit dem Bezeichner $value_2$ und $\boldsymbol{e(value_1)} = \{\, e \in E \mid label(e) = value_1 \,\}$ die Menge an Kanten mit dem Bezeichner $value_1$.

Ein $\boldsymbol{Pfad\ p}$ sei eine alternierende Folge von n Knoten und $n-1$ Kanten, die mit einem Startknoten $v_1 \in V$ beginnt und mit einem Endknoten $v_n \in V$ det, $n \in \mathbb{N}$. Sei weiterhin $\boldsymbol{P}$ eine Menge an Pfaden.

Mit einem $\boldsymbol{Lokationsschritt\ ls}$ wird eine Kante $e \in E$ spezifiziert, die ausgehend von einer Menge an Pfaden eine Menge an Knoten $v \in V$ selektiert. Formal ist die Bedeutung durch die nachfolgenden semantischen Funktionen $\boldsymbol{S}$ und $\boldsymbol{C}$ festgelegt.

Sei $\boldsymbol{lastnode(p)}$ der letzte Knoten $v_n \in V$ des Pfades p.

Und sei $\boldsymbol{extendPath(p, e, v)}$**:** Erzeugt einen neuen Pfad p', indem der Pfad p um die Kante e und den Knoten v erweitert wird, sodass gilt: $lastnode(p') = v$.

Und sei $\boldsymbol{createPath(v_1, v_2, e)}$: Erzeugt einen neuen Pfad p, der den Knoten v_1 über die Kante e mit den Endknoten v_2 verbindet.

$\mathcal{S}[\![ls]\!](P)$ steht für die Menge aller Pfade, die durch den Lokationsschritt ls aus der Pfadmenge P generiert werden.

Seien im Folgenden ls, ls_1, ls_2 Lokationsschritte, p, p_1, p_2 Pfade, P, P_1 Mengen an Pfaden, $i \in \mathbb{N} \setminus \{0\}$ und $node \in V$ ein (Start-)knoten.

$$\mathcal{S}: Lokationsschritt \rightarrow Menge\ an\ Pfaden \rightarrow Menge\ an\ Pfaden$$

$$\begin{aligned}
\mathcal{S}[\![\ ls_1 > ls_2\ ]\!](P) &= \mathcal{S}[\![ls_2]\!](\ \mathcal{S}[\![ls_1]\!](P)\ ) \\
\mathcal{S}[\![\ ls\ ]\!](P) &= \{\ p' = extendPath(p, e, v_2)\ \mid \\
&\quad\ p \in P, e = (v_1, v_2),\ \ label(e) = ls, \\
&\quad\ lastnode(p) = v_1,\ \ v_1, v_2 \in V\ \} \\
\mathcal{S}[\![\ node :: ls\ ]\!](P) &= \{\ p = createPath(v_1, v_2, e)\ \mid \\
&\quad\ e = (v_1, v_2),\ \ label(e) = ls, \\
&\quad\ label(v_1) = label(node),\ \ v_1, v_2 \in V\ \} \\
\mathcal{S}[\![\ ls(i)\ ]\!](P) &= if\ (i = 2)\ \ then\ \ \mathcal{S}[\![\ ls > ls\ ]\!](P) \\
&\quad\ else\ \ \mathcal{S}[\![\ ls > ls(i-1)\ ]\!](P) \\
\mathcal{S}[\![\ ls[cond]\ ]\!](P) &= Sei\ P_1 = \mathcal{S}[\![ls]\!](P)\ in \\
&\quad\ \{\ p_1 |\ p_1 \in P_1,\ \ \mathcal{C}[\![cond]\!](p_1, P_1)\ \}
\end{aligned}$$

$\mathcal{C}[\![cond]\!](p, P)$ steht für den booleschen Wert der die Bedingung $cond$, nachdem sie auf dem Pfad p und der Pfadmenge P ausgewertet wurde, $p \in P$.

Seien im Folgenden $cond, cond_1, cond_2$ Bedingungen, die einer der unten aufgeführten Funktionen entsprechen können, ls ein Lokationsschritt, p, p_1 Pfade, P, P_1, P_2 Mengen an Pfaden, $i \in \mathbb{Z}$, $v \in V$ und t eine beliebige Zeichenketten, die aus Buchstaben und Zahlen bestehen darf.

$$\mathcal{C}: Bedingung \rightarrow (Pfad, Menge\ an\ Pfaden) \rightarrow Boolean$$

$$\begin{aligned}
\mathcal{C}[\![[cond_1][cond_2]]\!](p, P) &= \mathcal{C}[\![cond_1]\!](p, P) \wedge \mathcal{C}[\![cond_2]\!](p, P) \\
\mathcal{C}[\![equals(t)]\!](p, P) &= equals(lastnode(p), t) \\
\mathcal{C}[\![prefix(t)]\!](p, P) &= prefix(lastnode(p), t) \\
\mathcal{C}[\![suffix(t)]\!](p, P) &= suffix(lastnode(p), t) \\
\mathcal{C}[\![min(i)]\!](p, P) &= min(lastnode(p), i) \\
\mathcal{C}[\![max(i)]\!](p, P) &= max(lastnode(p), i) \\
\mathcal{C}[\![ls = cond]\!](p, P) &= Sei\ P_1 = \mathcal{S}[\![ls]\!](P)\ \ in \\
&\quad\ if\ \{\ p_1 |\ p_1 \in P_1, \mathcal{C}[\![cond]\!](p_1, P_1)\ \} = \emptyset\ \ then\ \ false \\
&\quad\ else\ \ true
\end{aligned}$$

Seien dabei die Filterfunktionen wie folgt definiert:

$equals(v, t)$: true, falls $label(v) = t$

$prefix(v, t)$: true, falls $label(v)$ mit der Zeichenfolge t beginnt

$suffix(v, t)$: true, falls $label(v)$ mit der Zeichenfolge t endet

$min(v, i)$: true, falls $label(v) \geq i$

$max(v, i)$: true, falls $label(v) \leq i$

Weitere Details und Beispiele zu diesen fünf Filterfunktionen sind in Tabelle 2 aufgeführt, die im Rahmen der Syntax von RDFPath vorgestellt wurde.

3.5 Beispiele

Nachdem nun die einzelnen Komponenten von RDFPath eingeführt wurden, folgen in diesem Abschnitt einige ausführlichere Beispiele. Ziel ist es zum einen, die unterschiedlichen Ausdrucksmöglichkeiten problembezogen aufzuzeigen und zum anderen die einzelnen Auswertungsschritte einer Anfrage unter Verwendung von Graphen zu veranschaulichen.

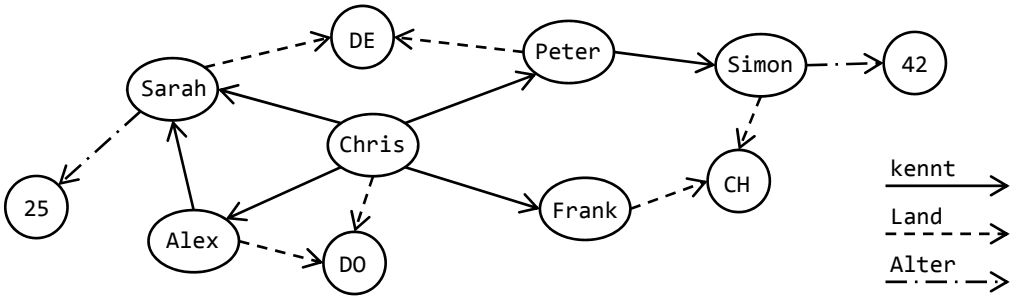
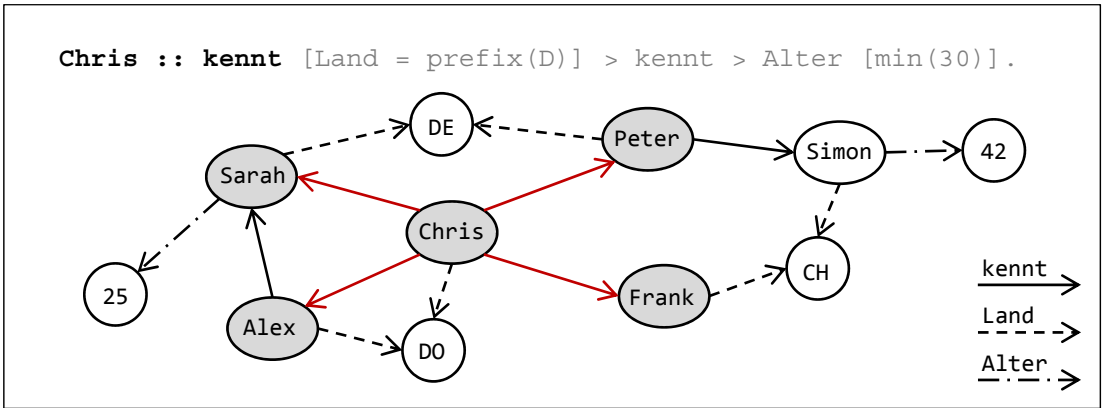


Abbildung 15: RDF-Graph mit Bekanntschaften

Abbildung 15 zeigt einen beispielhaften RDF-Graphen, der die Bekanntschaften mehrerer Personen modelliert. Hierbei gilt eine Person `X` mit einer anderen Person `Y` umgangssprachlich als befreundet, wenn der Graph eine durchgezogene Kante von `X` nach `Y` aufweist und das entsprechende RDF-Dokument folglich auch das Triple `(X, kennt, Y)` enthält. Weiterhin ordnet der Graph jeder Person ein Land zu (gestrichelte Kante) und zwei Personen ihr Alter (punktierte Kante). Da die Kanten in einem RDF-Graphen gerichtet sind, können Pfade nur entsprechend der Pfeilrichtung gebildet werden.

Beispiel 1

Im ersten Beispiel wird die Anwendung von Lokationsschritten, Unteranfragen sowie Filtern betrachtet. Die entsprechende Anfrage, der aktuell betrachtete Lokationsschritt, der dazugehörige Graph sowie die resultierenden Ergebnisse werden für jeden Auswertungsschritt in einer gesonderten Abbildung aufgeführt. Die diesem Beispiel zugrunde liegende Anfrage beginnt mit Abbildung 16, die den ersten Auswertungsschritt beschreibt. Zunächst sollen alle von „Chris" ausgehenden Knoten, die über die Kante „kennt" zu erreichen sind, betrachtet werden.



Zwischenergebnisse:

```
Chris (kennt) Peter
Chris (kennt) Alex
Chris (kennt) Sarah
Chris (kennt) Frank
```

Abbildung 16: Anfrage-Beispiel 1 - Auswertungsschritt (1/4)

Als nächstes werden unter Verwendung einer Unteranfrage alle Freunde von Chris verworfen, deren Land nicht mit dem Buchstaben „D" beginnt. Wie in Abbildung 17 dargestellt, wird die Bedingung der Unteranfrage von allen Knoten bis auf „Frank" erfüllt. Daher wird der Pfad von „Chris" nach „Frank" verworfen.

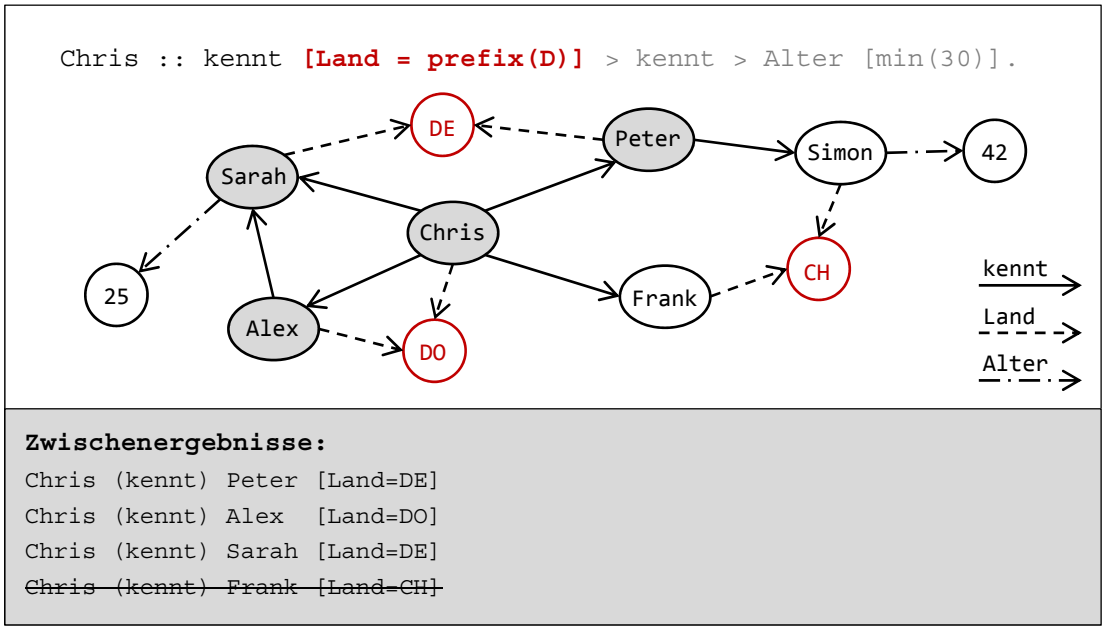


Abbildung 17: Anfrage-Beispiel 1 - Auswertungsschritt (2/4)

Im nächsten Lokationsschritt soll die Kante „kennt“ ein zweites Mal verfolgt werden. In Abbildung 18 ist zu sehen, dass der Knoten „Sarah“ über keine entsprechende Kante verfügt. Demnach wird der direkte Pfad von Chris nach Sarah nicht weiter betrachtet. Hingegen hat „Peter“ wie auch „Alex“ mindestens eine ausgehende „kennt“ Kante, sodass die beiden entsprechenden Pfade gemäß dem aktuellen Lokationsschritt erweitert werden. Falls einer der beiden Knoten mehr als nur eine ausgehende Kante hätte, würden zusätzliche Pfade in die Ergebnismenge aufgenommen werden.

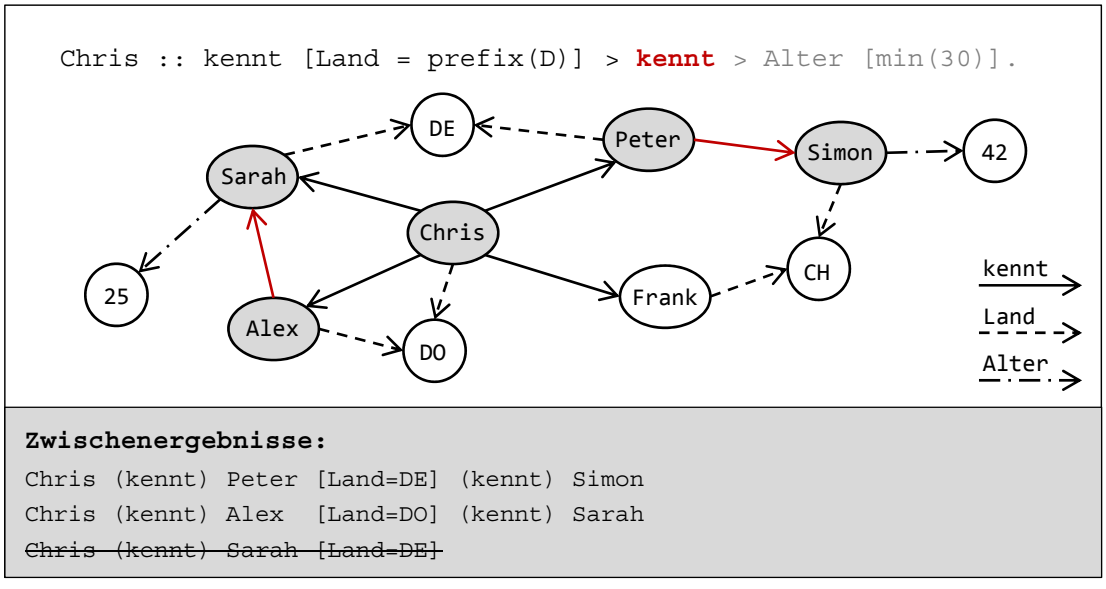


Abbildung 18: Anfrage-Beispiel 1 - Auswertungsschritt (3/4)

Im letzten Auswertungsschritt soll der Kante „Alter“ gefolgt werden, wobei nur diejenigen Knoten zulässig sind, deren Zahlenwert mindestens „30“ entspricht. Umgangssprachlich werden alle Personen, die jünger als 30 Jahre sind, verworfen. Wie Abbildung 19 zu entnehmen ist, trifft das nur noch auf einen der beiden verbliebenen Pfade zu. Dementsprechend liefert die Anfrage, auf diesem Graphen ausgeführt, lediglich einen Pfad zurück.

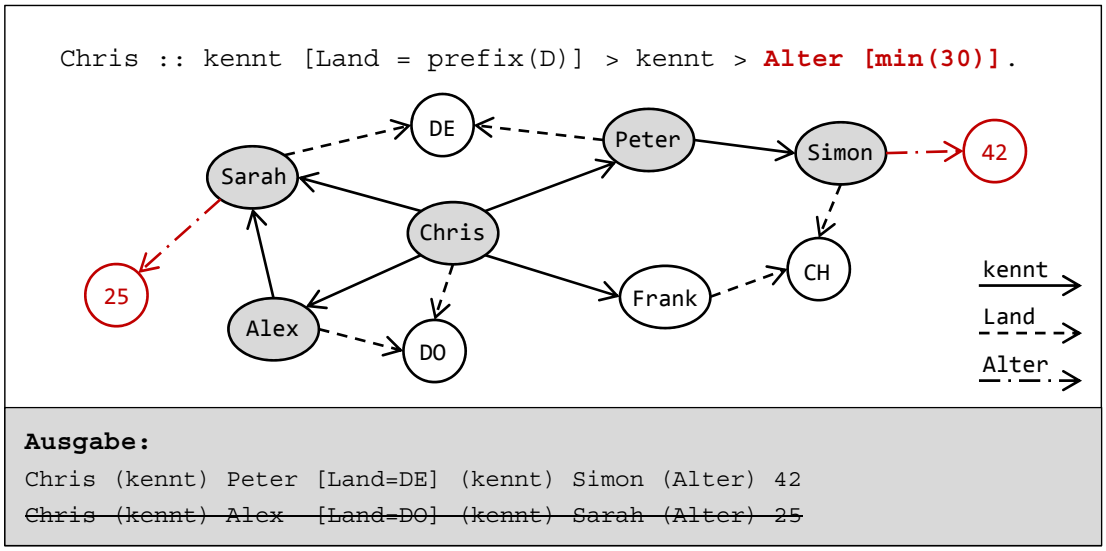


Abbildung 19: Anfrage-Beispiel 1 - Auswertungsschritt (4/4)

Beispiel 2

Im zweiten Beispiel wird auf dem gleichen RDF-Graphen eine Anfrage erläutert, die nach den kürzesten Pfaden sucht. Wie in Abbildung 20 dargestellt, werden zunächst alle von „Chris“ ausgehenden Kanten mit dem Bezeichner „kennt“ betrachtet.

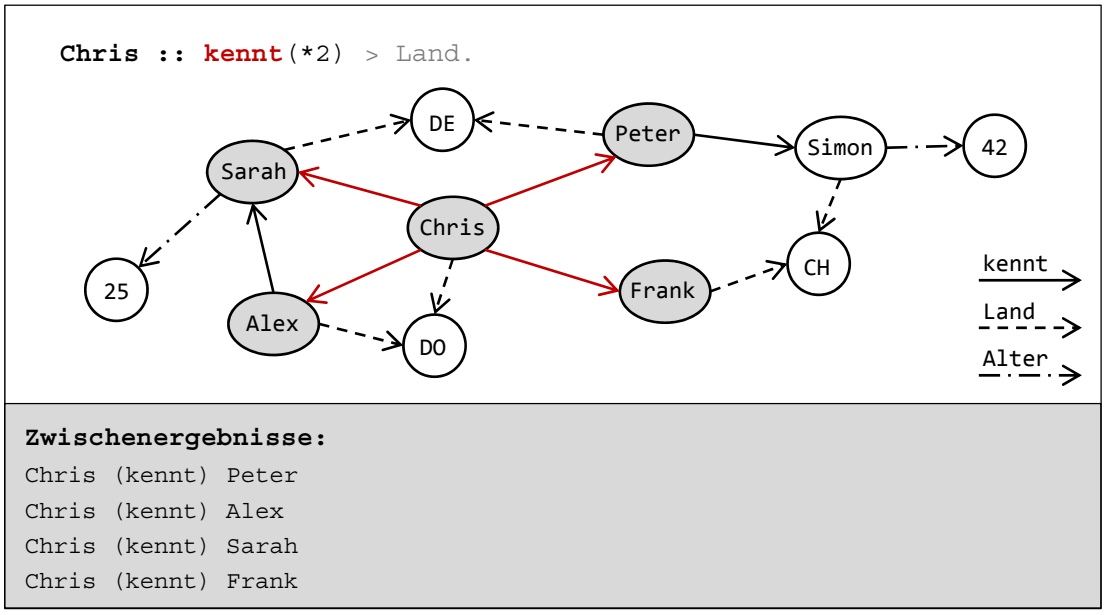
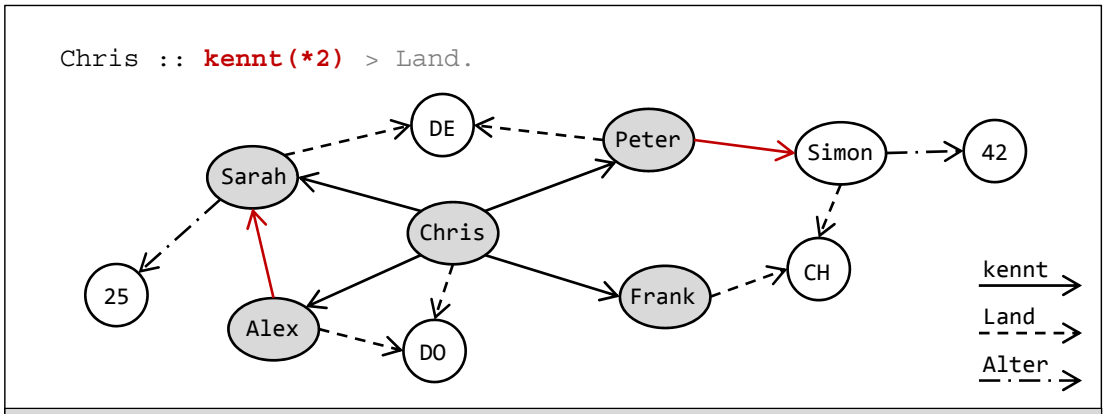


Abbildung 20: Anfrage-Beispiel 2 - Auswertungsschritt (1/3)

Als nächstes soll entsprechend der Anfrage ein weiteres Mal der Kante „kennt“ gefolgt werden. In Abbildung 21 wird gezeigt, dass hierdurch zum einen ein neuer Knoten „Simon“ erreicht wird und zum anderen ein bereits im vorherigen Lokationsschritt besuchter Knoten „Sarah“ über einen alternativen Pfad erreicht wird. Da nur nach den kürzesten Pfaden gesucht werden soll, wird der alternative Pfad, der um eine Kante länger ist, verworfen.

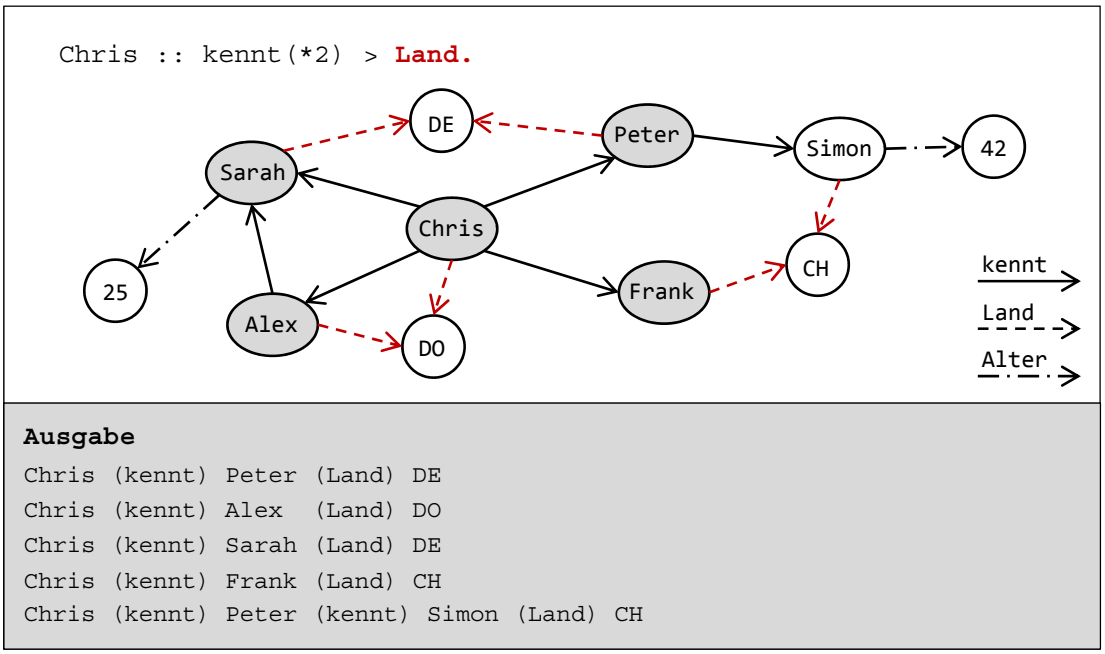


Zwischenergebnisse:

Chris (kennt) Peter
Chris (kennt) Alex
Chris (kennt) Sarah
Chris (kennt) Frank
Chris (kennt) Peter (kennt) Simon
~~Chris (kennt) Alex (kennt) Sarah~~

Abbildung 21: Anfrage-Beispiel 2 - Auswertungsschritt (2/3)

Im letzten Auswertungsschritt soll das Land betrachtet werden. Aus Abbildung 22 lässt sich entnehmen, dass bei allen betrachteten Pfaden die jeweils letzten Knoten über eine entsprechende Kante verfügen, sodass keine Pfade verworfen werden. Außerdem ist zu sehen, dass die unterschiedliche Länge der Pfade nicht von Bedeutung ist. Im Allgemeinen wird, unabhängig von der Pfadlänge, immer der letzte Knoten als Referenzknoten für weitere Lokationsschritte verwendet. Dementsprechend liefert die Anfrage, auf diesem Graphen ausgeführt, fünf Pfade unterschiedlicher Länge zurück.



Ausgabe

```
Chris (kennt) Peter (Land) DE
Chris (kennt) Alex  (Land) DO
Chris (kennt) Sarah (Land) DE
Chris (kennt) Frank (Land) CH
Chris (kennt) Peter (kennt) Simon (Land) CH
```

Abbildung 22: Anfrage-Beispiel 2 - Auswertungsschritt (3/3)

4 Implementierung

In diesem Kapitel wird die implementierte Überführung der zuvor eingeführten Pfadanfragesprache RDFPath in eine sequentielle Folge von MapReduce Aufträgen beschrieben. Hierfür wird zunächst ein konzeptueller Überblick über die einzelnen Komponenten dargelegt und im Anschluss daran die wesentlichen Techniken ausführlich erläutert.

Die im Rahmen dieser Arbeit entwickelte Implementierung wurde, wie das Hadoop MapReduce-Framework, in Java programmiert. Bevor nun die einzelnen Bestandteile besprochen werden, liefert Abbildung 23 zunächst einmal einen Überblick über das Zusammenspiel und den Umfang der implementierten Komponenten. Als zusätzliche Orientierungshilfe wurden die jeweiligen Punkte mit einem Verweis auf das entsprechende Kapitel versehen.

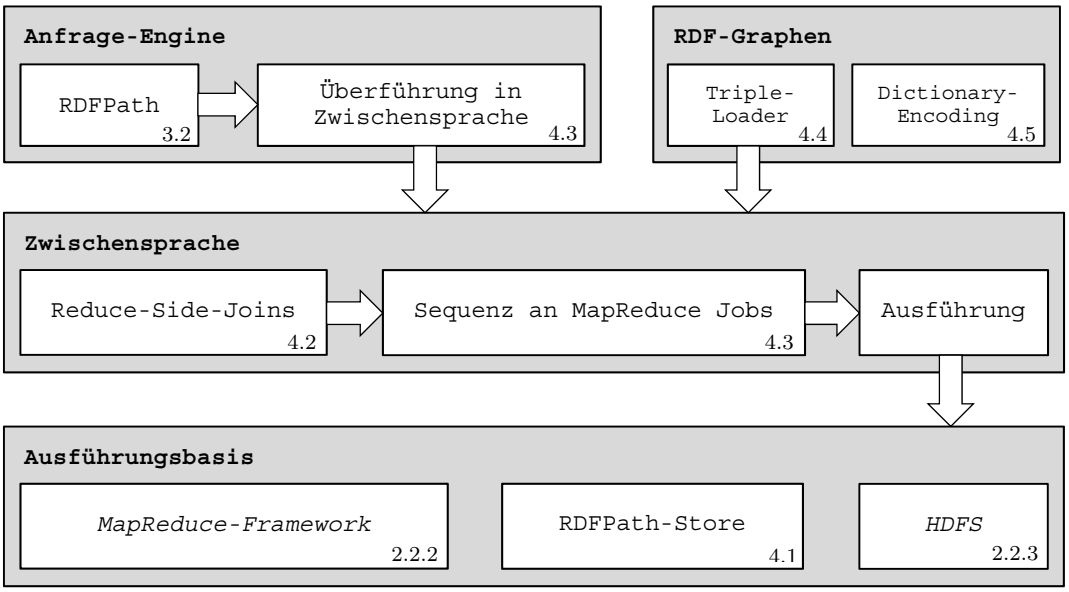


Abbildung 23: Überblick der Implementierung

Im Folgenden wird das Aufgabenfeld der vier übergeordneten Bereiche (graue Markierung) kurz aufgeführt.

- **Anfrage-Engine:** Die RDFPath-Anfrage wird über eine textbasierte Eingabeaufforderung vom *System* entgegengenommen, auf Fehler überprüft und in einer *Zwischensprache* ausgedrückt.
- **Zwischensprache:** Unter Verwendung der Zwischensprache können *Joins*, Filter und Unteranfragen spezifiziert werden, die wiederum in eine Folge an MapReduce Aufträgen überführt werden.

- **Ausführungsbasis:** Die Folge von MapReduce Aufträgen wird im MapReduce-Framework ausgeführt, wobei auf den *RDFPath-Store*, der im HDFS abgelegt ist, zugegriffen wird.
- **RDF-Graphen:** Ein neues RDF-Dokument wird in den *RDFPath-Store* eingepflegt. Optional kann hierbei dass in Kapitel 4.5 spezifizierte *Dictionary-Encoding* angewendet werden. Die hierfür notwendigen Operationen werden ebenfalls in der Zwischensprache modelliert.

Das restliche Kapitel ist wie folgt aufgebaut. Zunächst wird mit dem *RDF-Path-Store* ein Datenmodell vorgestellt, auf dem Anfragen arbeiten können. Anschließend werden *Reduce-Side-Joins* eingeführt und ein Zusammenhang zu der Auswertung einer Anfrage sowie zum Datenmodell hergestellt. Darauf aufbauend wird ein technischer Blick auf den konkreten *Überführungsprozess* einer RDFPath Anfrage in eine Sequenz an MapReduce Aufträgen geworfen. Abschließend werden noch der *Triple-Loader* und das *Dictionary Encoding* besprochen.

4.1 RDFPath-Store

Unter dem Begriff *RDFPath-Store* werden in dieser Arbeit Konzepte zur Speicherung von RDF-Dokumenten zusammengefasst. Demgemäß wird in diesem Abschnitt das Datenmodell spezifiziert, auf dessen Grundlage die Anfragen ausgewertet werden. Hierfür werden zunächst die Anforderungen an ein solches Datenmodell dargelegt und im Anschluss daran Alternativen und Möglichkeiten in einem Hadoop System besprochen. Zuletzt werden das im Rahmen dieser Arbeit entwickelte *Speicherkonzept* und die gewählten Strategien vorgestellt.

Die Ausgangssituation zur Konzeption eines Datenmodells sah die Speicherung von RDF-Dokumenten vor, die zwischen einer Million und mehreren Milliarden RDF-Triples[22] enthalten konnten. Ziel war es dabei Strategien zu entwickeln, die eine möglichst effiziente Auswertung von Pfadanfragen unterstützen. Als Grundlage hierfür sollten bereits etablierte Systeme wie das Hadoop Distributed Filesystem, HBase oder die Anbindung lokaler Ressourcen Verwendung finden. Eine der wesentlichsten Anforderungen an ein solches System lag dabei in der *verteilten Speicherung* der RDF-Graphen. So sollte hier vor allem eine faire und robuste Verteilung der Daten auf alle Maschinen dafür

[22] Die Größe von 1 Mio. Triples kann sehr stark variieren, einige Beispiele sind in Kapitel 6 Evaluation nachzulesen.

Sorge tragen, dass verteilte Berechnungen die benötigten RDF-Graphen möglichst schnell einlesen können.

Weiterhin sollte die Menge an Daten, die für eine Berechnung betrachtet werden muss, möglichst gering ausfallen. Infolgedessen sollte eine Anfrage, die nur auf einem kleinen Untergraphen des RDF-Dokuments arbeitet, auch möglichst nur diesen kleinen Untergraphen einzulesen brauchen. Das vollständige Betrachten eines RDF-Dokumentes kann allerdings nur dann vermieden werden, wenn eine *Selektion* benötigter Teildaten vorgenommen werden kann. Dies setzt wiederrum voraus, dass die Daten eines RDF-Dokuments im Vorfeld strukturiert oder sortiert worden sind, sodass unter Verwendung eines *Indexes* oder einer *Datenpartitionierung* direkt auf die entsprechenden Datenteile zugegriffen werden kann. Abschließend war es noch von entscheidender Bedeutung, dass sich die gewählten Systeme und Strategien durch eine hohe Flexibilität und Kompatibilität auszeichneten. So sollten zum einen weitere Optimierungsstrategien anwendbar bleiben und zum anderen gewährleistet werden, dass die Verarbeitungsprinzipien des MapReduce Frameworks vorteilhaft ausgenutzt werden können.

Entsprechend diesen Anforderungen kamen als Grundsystem nur noch *HBase* und das *Hadoop Distributed Filesystem* in Frage. *HBase* kann, wie bereits in Kapitel 2.2.1 erläutert, als verteiltes Speichersystem für MapReduce Aufträge eingesetzt werden. Dabei werden die gespeicherten Daten auf die im Rechencluster zur Verfügung stehenden Rechner verteilt und können somit parallel gelesen und verarbeitet werden. Auch die spaltenbasierte Struktur sowie die auf *Random-Access* ausgelegten Zugriffsmechanismen werden der geforderten Möglichkeit, nur benötigte Teildaten selektieren zu können, gerecht.

Auf der contra Seite muss bedacht werden, dass HBase zwar in das MapReduce Framework eingebettet ist, jedoch auf eine eigenständige Architektur zurückgreift. Diese erfordert zum einen zusätzliche hardwareseitige Ressourcen und zum anderen die Administration weiterer softwareseitiger Komponenten. Ein weiterer Kritikpunkt betrifft die Flexibilität dieses Systems. HBase wird als ein abgeschlossenes System verwendet, das abzuspeichernde Daten entgegennehmen und angefragte Daten zurückgeben kann. Die Implementierung von problemspezifischen Optimierungsstrategien könnte folglich umständlich zu realisieren sein. Der ausschlaggebende Kritikpunkt jedoch war die zum Zeitpunkt der Entscheidungsfindung relativ frühe Entwicklungsphase. So wurde

beispielsweise in [White, 2009][23] über Komplikationen im Zusammenspiel mit dem Hadoop Distributed Filesystem berichtet.

Diese und weitere kritische Anmerkungen zu aktuellen Versionen führten dazu, dass im Rahmen dieser Arbeit von einer weiteren Betrachtung von HBase abgesehen wurde. Für zukünftige Projekte wäre allerdings eine erneute Beurteilung zu erwägen, da die Weiterentwicklung, insbesondere durch Facebook, sehr stark vorangetrieben wird und die Vorschau auf geplante Neuerungen sehr vielversprechend wirkt[24]. Dessen ungeachtet fiel die die Entscheidung fiel zu Gunsten des Hadoop Distributed Filesystems aus.

Das *Hadoop Distributed Filesystem (HDFS)* erfüllt mit den in Kapitel 2.2.3 besprochenen Merkmalen alle hier festgelegten Anforderungen. Die Daten werden in Blöcke unterteilt, automatisch auf dem gesamten Rechencluster abgespeichert und können parallel verarbeitet werden. Da wie bei einem gewöhnlichen Dateisystem hierarchische Ordnerstrukturen und beliebige Dateibezeichner unterstützt werden, ist eine Partitionierung der Daten prinzipiell möglich und somit entsprechende Strategien, welche die zu verarbeitende Datenmenge durch Selektion reduzieren, realisierbar. Die Punkte Kompatibilität und Zuverlässigkeit wurden bereits in den Grundlagen dieser Ausarbeitung abgehandelt und positiv beurteilt.

4.1.1 Speicherkonzept

Ein *naives Speicherkonzept* könnte vorsehen, RDF-Dokumente als Ganzes in das Hadoop Distributed Filesystem zu übertragen. Das RDF-Dokument würde dann auf alle zur Verfügung stehenden Datanodes verteilt und das MapReduce-Framework könnte die Berechnungen automatisch parallelisieren. Eine Analyse von RDF-Graphen wäre mit der zuvor spezifizierten Anfragesprache RDFPath ohne größere Einschränkungen durchführbar. Ausgehend von diesem trivialen Ansatz, der keinerlei Optimierungen hinsichtlich der Ausführungszeit aufweist, werden im Folgenden mehrere Konzepte vorgestellt, welche das Ziel verfolgen, Pfadanfragen im MapReduce Framework möglichst effizient auszuwerten.

Eine wichtige Einflussgröße stellt in diesem Zusammenhang der Algorithmus, der die Blöcke im HDFS auf Datanodes verteilt, dar. Nach [Cloudera, 2010] wurde dieser hinsichtlich folgender Eigenschaften optimiert: *hinreichende Redundanz*, *hohe Schreib-/Lese-Bandbreite* sowie *gleichmäßige Verteilung* im

[23] Chapter 12, Love and Hate: HBase and HDFS

[24] http://wiki.apache.org/hadoop/Hbase/HBaseVersions

Cluster. Es hat sich gezeigt, dass die hier zugrunde liegenden Algorithmen fest in den Quellcode von HDFS eingebunden sind und Optimierungen folglich weitreichender Änderungen bedürften. Dies war im Rahmen dieser Arbeit jedoch nicht möglich. Dementsprechend sollten sich Optimierungen auf die Abbildung der Daten im HDFS konzentrieren und, wie zuvor motiviert, beispielsweise eine *Partitionierung* von RDF-Dokumenten anstreben. Eine wesentliche Überlegung hierbei war, dass Lokationsschritte immer nur eine Kante und somit nur einen ganz spezifischen Teil der Daten betrachten. Des Weiteren können Kanten (Prädikate) in RDFPath – im Gegensatz zu SPARQL – niemals *Variablen* sein. Sie sind immer fest vorgeschrieben. Demzufolge wäre es vorteilhaft, nur diejenigen Triples eines RDF-Dokuments lesen zu können, die das entsprechende Prädikat aufweisen. Dieser Gedankengang begründet die implementierte Strategie, Triples eines RDF-Dokumentes im Vorfeld nach ihren Prädikaten zu unterteilen.

Dieses, auch als *vertikale Partitionierung* [Abadi, et al., 2007] bekannte, Konzept bildet die Grundlage für das Datenmodell der Implementierung. In Abbildung 24 ist die vertikale Partitionierung des aus den vorherigen Beispielen bekannten RDF-Graphen dargestellt. Wie auf der Abbildung zu erkennen ist, wird im HDFS für jede neue Kante eine weitere Datei angelegt und alle gleich bezeichneten Kanten in einer zusammenhängenden Datei abgespeichert. Die dabei genutzten Datenstrukturen und das interne Format einer solchen Datei, die im Folgenden als `SequenceFile` eingeführt wird, werden im nächsten Abschnitt weiter spezifiziert.

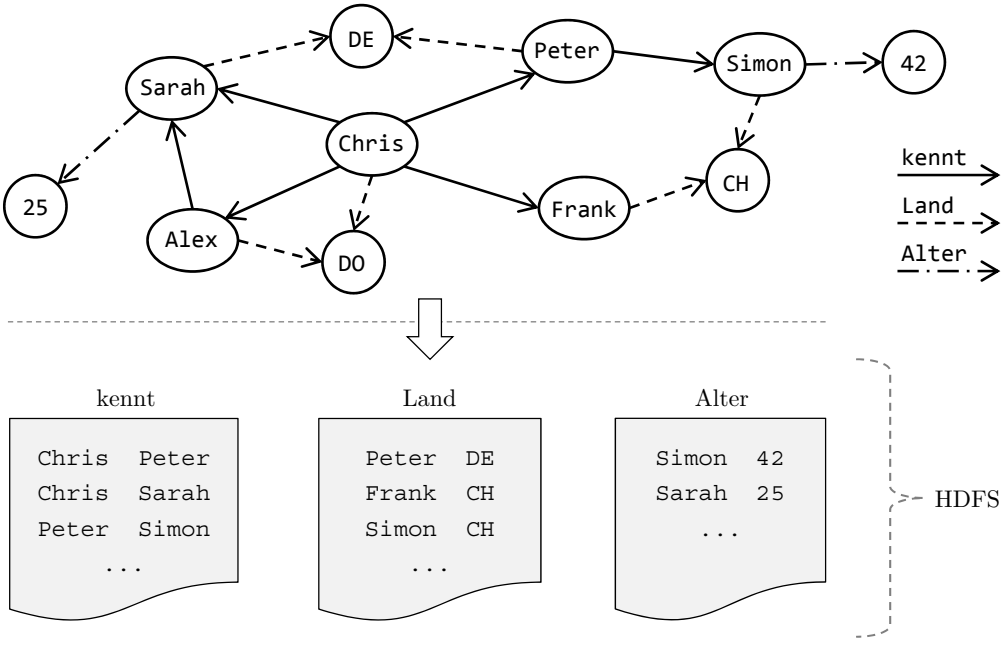


Abbildung 24: Vertikale Partitionierung eines RDF-Graphen

4.1.2 Typen

Im Kontext des Datenmodells ist es insbesondere für das Verständnis der im nachfolgenden Kapitel besprochenen *Reduce-Side-Joins* sinnvoll, die drei wesentlichen Datenstrukturen `PathWritable`, `TextPairWritable` und `IntPairWritable` im Vorfeld einzuführen und das Format, in dem diese Datenstrukturen abgelegt werden, genauer zu betrachten.

Das Hadoop Distributed Filesystem sieht innerhalb einer Datei standardmäßig eine *zeilenbasierte* Trennung unterschiedlicher Einträge vor. Der Inhalt einer jeden Zeile wird dabei im Regelfall dem Objekttyp `Text` zugeordnet. Hierbei handelt es sich um ein Äquivalent zu dem aus Java bekannten `java.lang.String`-Objekt. Neben dem Objekttyp `Text` gibt es in Hadoop noch weitere spezielle *Wrapper*-Klassen[25], zu denen beispielsweise `BooleanWritable, ByteWritable, IntWritable` sowie `DoubleWritable` zählen. Die jeweiligen *Java Primitive* sind den Bezeichnern zu entnehmen. Zu den wesentlichen Änderungen zählt zum einen der für MapReduce optimierte *Serialisierungsprozess*. Hierbei werden strukturierte Objekte in einen Byte-Stream umgewandelt, um entweder direkt abgespeichert oder über das Netzwerk transferiert zu werden. Zum anderen sind die auch als *Writables* bezeichneten Objekte darauf ausgelegt, ein kompaktes und vor allem auch schnelles, persistentes *Speicherformat* bereitzustellen, sodass der Speicherplatz möglichst effektiv ausgenutzt werden kann und der *Overhead* für das Lesen und Schreiben von mehreren Terabytes an Daten möglichst gering ausfällt. Eine umfassendere Betrachtung der zur Verfügung stehenden Writables ist im Rahmen dieser Arbeit nicht möglich. Für eine detaillierte Einführung ist [White, 2009], Kapitel 4 zu empfehlen.

Auch wenn die von Hadoop zur Verfügung gestellten Writable-Implementierungen für viele Anwendungsbereiche ausreichen, so kann die Implementierung eines eigenen *Writable Objektes* bei datenintensiven Vorgängen signifikanten Einfluss auf die Effizienz des gesamten Auswertungsprozesses haben. Im Folgenden werden nun drei wesentliche Writable Implementierungen vorgestellt, die ihm Zuge dieser Arbeit entwickelt wurden. Abschließend wird noch ein kurzer Blick auf den verwendeten Dateityp geworfen.

PathWritable

Der Objekttyp `PathWritable` wurde für die Speicherung und Modellierung kompletter Pfade im MapReduce-Framework entworfen. Dementsprechend repräsentiert dieses Objekt Informationen über eine Folge an Knoten und Kanten

[25] Objektstruktur, die einen primitiven Datentyp in ein Objekt aufnimmt.

die ausgehend vom Startknoten betrachtet wurde. Bei der internen Speicherung dieser Informationen wurden möglichst speichereffiziente Datenstrukturen verwendet, die bereits von Hadoop hinsichtlich Serialisierbarkeit und Kompaktheit optimiert wurden. Folglich stellt ein `PathWritable` Objekt eine Komposition mehrerer zuvor vorgestellter Writables dar. Weiterhin benötigt das Modell eines Pfades zahlreiche *Interaktionsmöglichkeiten*, um beispielsweise einen weiteren Knoten hinzuzufügen oder seine aktuelle Länge zu bestimmen. Die wichtigsten Methoden können dem Klassendiagramm in Abbildung 25 entnommen werden.

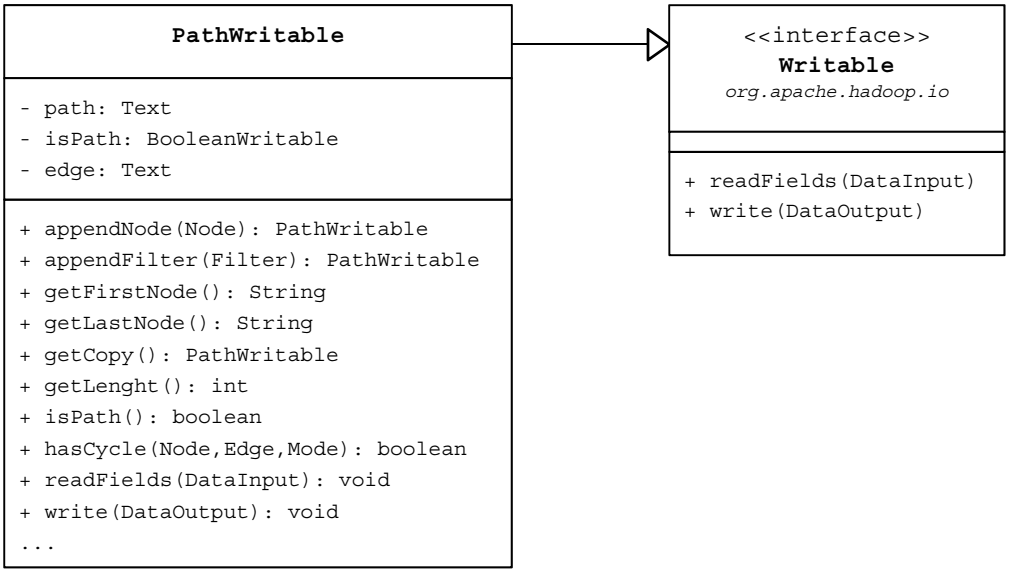


Abbildung 25: Klassendiagramm PathWritable

TextPairWritable

Der Objekttyp `TextPairWritable` wurde für die später folgenden *Reduce-Side-Joins* entworfen. Er setzt sich aus zwei `Text`-Objekten zusammen und wurde einem Beispiel in [White, 2009] nachempfunden. Da diese *Objekt-Komposition* für die Repräsentation von Schlüsselwerten verwendet wird, die bei der Auswertung von Anfragen eine zentrale Rolle einnehmen, ist hier insbesondere auf eine effiziente Implementierung von *Vergleichsoperatoren*[26] zu achten. In diesem Zusammenhang wurden Methoden implementiert, die es ermöglichen `TextPairWritable` Objekte in ihrer *Byte-Repräsentation* miteinander zu vergleichen, ohne dass eine *Deserialisierung* in strukturierte Objekte durchgeführt werden muss.

[26] Entsprechen den aus Java bekannten `compareTo()` Methoden.

IntPairWritable

Bei diesem Objekt handelt es sich um ein Integer Äquivalent zum `TextPairWritable`. Es wird ebenfalls im Rahmen der Reduce-Side-Joins für die Repräsentation von Schlüsselwerten verwendet und kann demzufolge auch auf effiziente Vergleichsoperatoren zurückgreifen, die auf Byte-Ebene arbeiten.

SequenceFile

Wie bereits zuvor erläutert, verfolgt Hadoop innerhalb von Dateien standardmäßig einen zeilenorientierten Ansatz, wobei jede Zeile als Schlüssel-/Wertpaar interpretiert wird. Dabei darf sowohl der Schlüssel als auch der Wert eine beliebige Writable-Implementierung darstellen. An dieser Stelle kommt jedoch eine im Kapitel 4.3 beschriebene Problematik mit der neuen MapReduce-API zum Tragen. Mehrere Klassen waren zum Zeitpunkt der Implementierung noch nicht vollständig in die neue *MapReduce API* portiert. Dies führte dazu, dass `PathWritable`-Objekte zwar innerhalb eines MapReduce Auftrages in Form von Zwischenergebnissen verwendet werden konnten, eine persistente Speicherung für nachfolgende MapReduce Aufträge war allerdings nicht möglich. Entsprechend dem zuvor vorgestellten Speicherkonzept musste jedoch im Vorfeld etwaiger Analysen eine Partitionierung des RDF-Dokumentes durchgeführt werden. Hierbei sollten die `PathWritable`-Objekte persistent für zukünftige MapReduce Aufträge abgespeichert werden (siehe auch Kapitel 4.4: TripleLoader).

Eine Lösung wurde durch den Verzicht auf den zeilenorientierten Ansatz und die Verwendung von *dateibasierten Containern* gefunden. Diese beschreiben besondere Datenstrukturen in Hadoop, die darauf ausgelegt sind, komplexere Objekte aufzunehmen. Die einfachste Form dieser Container, die auch für die Speicherung von `PathWritable`-Objekten eingesetzt wurde, bildet die sogenannte `SequenceFile`. Ihre interne Struktur setzt sich aus einem Header, der Informationen über die enthaltenen Objekttypen bereitstellt und einer Sequenz an Schlüssel-/Wertpaaren zusammen. Der Speicherplatz-Overhead soll dabei weniger als einen Prozent betragen [White, 2009]. Es hat sich gezeigt, dass diese Datenstruktur sowohl für die persistente Speicherung von Pfaden, als auch für Zwischenergebnisse eine geeignete Lösung darstellt.

4.2 Reduce-Side-Joins

Im MapReduce-Framework ist es möglich, *Joins* zwischen großen Datenbeständen durchzuführen. Hierbei werden, wie in *relationalen Datenbanken* üblich, zwei Mengen unter Angabe eines gemeinsamen Schlüssels zu einer einzigen Menge zusammengeführt. Je nachdem wie groß diese Mengen bzw. Datenbestände sind, gibt es in MapReduce mehrere Strategien um Joins zu implementieren. Wenn zum Beispiel einer der beiden Datenbestände klein genug ist, um auf jeden Rechner im Hadoop-Cluster kopiert zu werden, würde es ausreichen, wenn ein MapReduce Auftrag lediglich eine partielle Sortierung der größeren Datenmenge nach dem *Join-Schlüssel* durchführt. Für jeden betrachteten Eintrag könnten die für den Join erforderlichen Informationen somit in der kleineren Datenmenge lokal nachgeschlagen werden. Falls beide Datenbestände zu groß sind, um auf alle Rechner kopiert zu werden, können sie mit Hilfe von *Map-Side-* oder *Reduce-Side-Joins* dennoch zusammengeführt werden. Bei der Anwendung von *Map-Side-Joins* ist allerdings zu beachten, dass strikte Voraussetzungen hinsichtlich der *Partitionierung* und *Sortierung* der Datenbestände erfüllt sein müssen. Aus diesem Grund wurden im Rahmen dieser Arbeit Reduce-Side-Joins implementiert, die flexibler eingesetzt werden können und keine strukturellen Bedingungen an die Datenbestände knüpfen. Im Gegensatz zu den Map-Side-Joins müssen hier allerdings mehr Daten über das Netzwerk übertragen werden.

Im Folgenden werden zunächst die allgemeinen Grundlagen eines Reduce-Side-Joins erörtert. Im Anschluss daran wird der Bezug zur eigentlichen Problemstellung – Lokationsschritte auf dem zuvor eingeführten RDFPath-Store auszuwerten – hergestellt und die Ausführung eines solchen Joins grafisch dargestellt.

Grundlagen

Bei einem *Reduce-Side-Join* werden in der Map-Phase zunächst beide Datenbestände gelesen und erwartungsgemäß neue Schlüssel-/Wertpaare erzeugt. Der Wert setzt sich dabei aus der eingelesenen Zeile oder einem Fragment davon zusammen. Für den Schlüssel wird nun der gewünschte *Join-Schlüssel* aus der aktuellen Zeile abgeleitet. Das hat zur Folge, dass in der darauffolgenden Reduce-Phase alle Schlüssel-/Wertpaare mit dem gleichen Join-Schlüssel in der gleichen Reduce-Funktion zusammengebracht werden. Demgemäß wird der Join zwischen den beiden Datenbeständen in den Reduce-Funktionen, die im Rechencluster verteilt aufgerufen werden, berechnet.

Da bei der Berechnung eines Joins eine Unterscheidung zwischen linkem und rechtem Datenbestand von entscheidender Bedeutung sein kann, kommen bei einem Reduce-Side-Join *zusammengesetzte Schlüssel* zum Einsatz. Sie wurden in Kapitel 4.1.2 als `TextPairWritable` und `IntPairWritable` vorgestellt. Folglich erhalten die Map-Funktionen eine zusätzliche Aufgabe, die darin besteht, den Join-Schlüssel um einen weiteren Schlüssel zu ergänzen. Dieser kennzeichnet den linken und den rechten Datenbestand, sodass innerhalb einer Reduce-Funktion beide unterschieden werden können. Die Verwendung von zusammengesetzten Schlüsseln hat jedoch weitreichende Auswirkungen auf die *Partitionierung* und damit auch auf die *Verteilung* der Schlüssel-/Wertpaare. Der hierfür zuständige Partitionierer, der diejenigen Paare, die den gleichen Schlüsselwert haben, der gleichen Reduce-Funktion zuordnet, darf nicht den zusammengesetzten Schlüssel betrachten, sondern nur den Join-Schlüssel.

Hingegen ist es für die Sortierung der Einträge, die als zusammenhängende Listen an Reduce-Funktionen übergeben werden, erforderlich auch die zusätzlichen Schlüssel zu betrachten. Nur so wird gewährleistet, dass zunächst alle Einträge des rechten und danach erst alle Einträge des linken Datenbestandes (oder umgekehrt) verarbeitet werden. Diese strukturelle Eigenschaft ist von entscheidender Bedeutung, da bei der Berechnung eines Reduce-Side-Joins eine der beiden Seiten zunächst zwischengespeichert werden muss.

Berechnung eines Lokationsschrittes

Diese Reduce-Side-Joins beschreiben nun die eigentliche Kernkomponente eines Lokationsschrittes. Für jede Betrachtung einer weiteren Kante ist ein Join zwischen der Menge an bisher berechneten Pfaden und der Datenpartition mit der entsprechenden Kante erforderlich. Dieser Zusammenhang ist in Abbildung 26 grafisch dargestellt. Als Grundlage wurde das in Kapitel 2.2.2 vorgestellte Beispiel 2 verwendet.

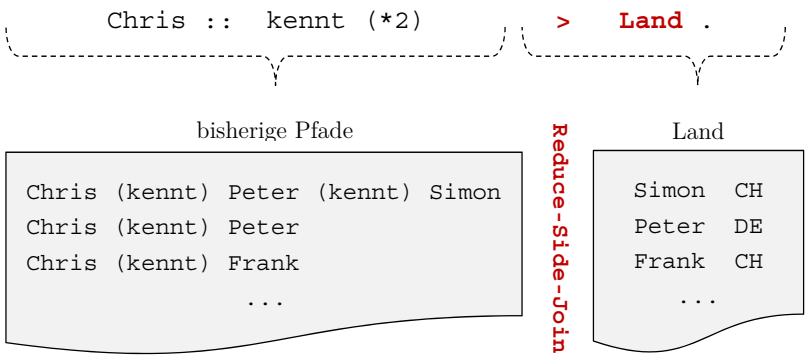


Abbildung 26: Reduce-Side-Joins und Lokationsschritte

Als Join-Schlüssel werden dabei, wie in der nächsten Abbildung 27 zu erkennen ist, zum einen die letzten Knoten der bisher berechneten Pfade und zum anderen die ersten Knoten der Datenpartition verwendet. Der zusätzliche Schlüssel ist in Abhängigkeit des aktuell betrachteten Datenbestandes „0“ oder „1“.

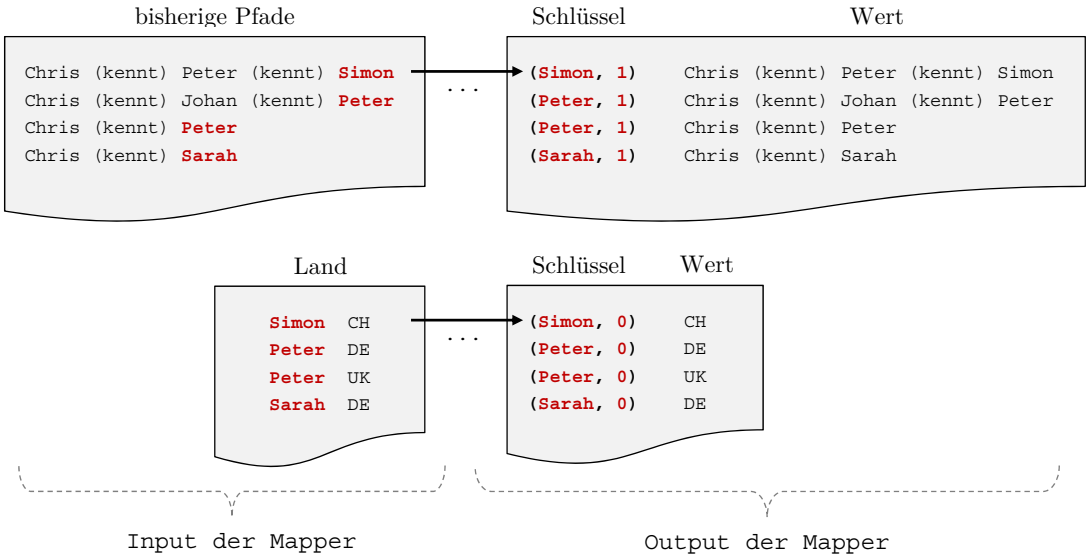


Abbildung 27: Reduce-Side-Join (Map-Phase)

In der darauffolgenden Shuffle-Phase werden nun alle Einträge bezüglich der ersten Schlüsselhälfte, im aktuellen Beispiel wäre das der Name, partitioniert. Dementsprechend werden alle Schlüssel-/Wertpaare mit der gleichen ersten Schlüsselhälfte – unabhängig aus welchem Datenbestand sie stammen – einer gemeinsamen *Partition* zugeordnet. Innerhalb einer solchen *Partition* werden die Einträge nach dem ganzen Schlüssel sortiert, sodass zunächst alle Einträge aus der Datenpartition und danach erst die bisherigen Pfade kommen.

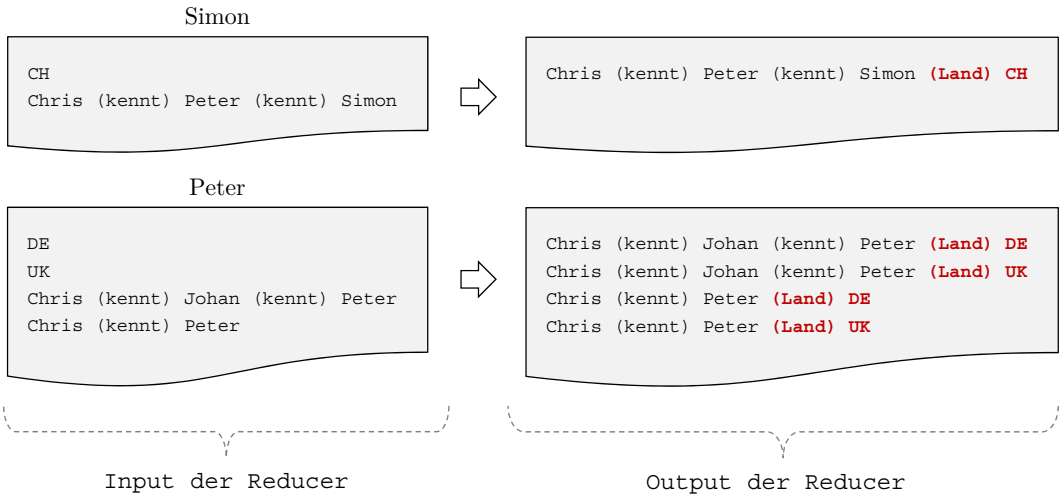


Abbildung 28: Reduce-Side-Join (Reduce-Phase)

In Abbildung 28 sind zwei in der Shuffle-Phase entstandene Partitionen dargestellt, die im nächsten Schritt von Reduce-Methoden verarbeitet werden. Wie zu erkennen ist, wird für den Knoten „Simon“ zunächst das Land aufgeführt und danach die bisherigen Pfade. Der allgemeinere Fall, nämlich dass ein Knoten mehrere ausgehende Kanten mit dem gleichen Bezeichner aufweist, wurde exemplarisch für den Knoten „Peter“ und die ausgehende Kante „Land“ konstruiert, indem „Peter“ zwei Länder „DE“ und „UK“ zugeordnet wurden. Die implementierten Reduce-Methoden speichern sich folglich zunächst alle Knoten ab, die ausgehend vom aktuell betrachteten Knoten über den spezifizierten Lokationsschritt erreicht werden können. Abschließend wird die restliche Liste durchlaufen, die nur aus Pfaden bestehen darf. Dabei wird jeder Pfad mit allen zwischengespeicherten Knoten zusammengeführt.

4.3 RDFPath in MapReduce

Nachdem nun die Anfragesprache, der RDFPath-Store und das Konzept des Reduce-Side-Joins aufgezeigt wurden, kann nun in diesem Abschnitt ein detaillierter Einblick in die konkrete Überführung einer in RDFPath ausgedrückten Anfrage in eine Sequenz an MapReduce Aufträgen gegeben werden.

Ausgehend von einer Anfrage, die über eine textbasierte Eingabeaufforderung vom System entgegengenommen wurde, wird zunächst der *Query-Parser* aufgerufen. Dieser und alle weiteren Schritte sind grafisch in der Abbildung 29 dargestellt. Der *Query-Parser* kontrolliert die erhaltene Anfrage auf syntaktische Fehler und das Vorhandensein der zu betrachtenden Kanten im RDFPath-Store. Anschließend wird die Anfrage in ihre Lokationsschritte aufgeteilt. Bei einer höheren Sprache, wie beispielsweise SPARQL, müssten an dieser Stelle nun *Syntax-Bäume* erzeugt und in eine Form transformiert werden, die eine sequentielle MapReduce Auswertung gestattet [Schätzle, 2010]. Wie im vorherigen Kapitel erläutert, wurden die mit RDFPath spezifizierten Lokationsschritte allerdings so konzipiert, dass sie direkt auf Reduce-Side-Joins abgebildet werden können. Hierfür werden die einzelnen Lokationsschritte zunächst in abstrakten *Instruktionen* ausgedrückt, die mehrere Anweisungen eines Lokationsschrittes, wie beispielsweise Reduce-Side-Joins, Filter oder Unteranfragen, zusammenfassen. Da die Instruktionen eine feste Abarbeitungsreihenfolge besitzen, werden sie in einer *Instruktions-Sequenz* zusammengefasst.

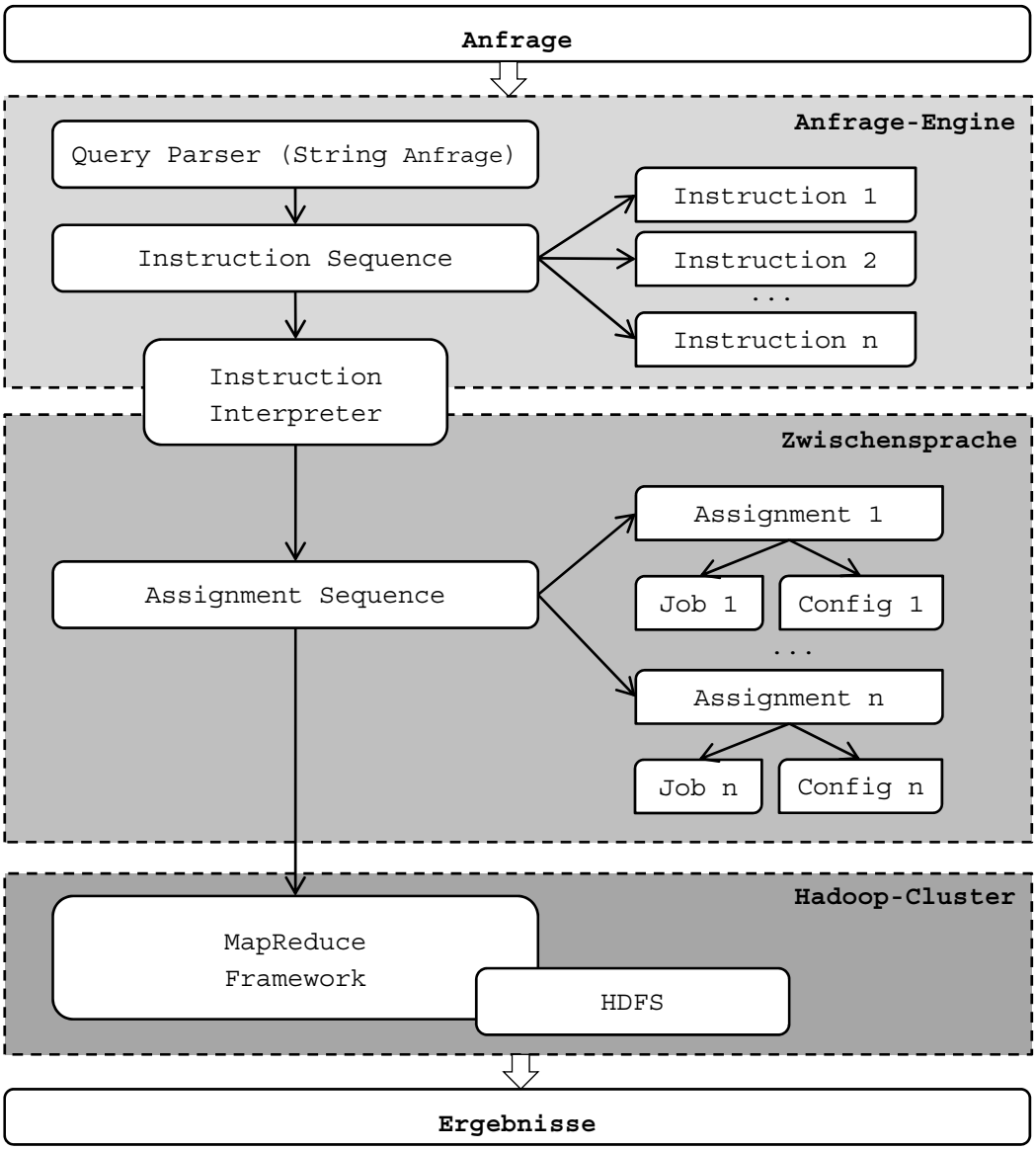


Abbildung 29: Schematischer Ablauf einer Anfrage

Nachdem die komplette Anfrage auf eine *Instruktions-Sequenz* abgebildet wurde, wird diese im nächsten Schritt dem *Instruktions-Interpreter* übergeben. Seine Aufgabe besteht darin, Instruktions-Sequenzen in sogenannte *Assignment-Sequenzen* zu überführen. Während *Instruktionen* abstrakte Objekte modellieren, die einfach nur mehrere Anweisungen zusammenfassen, handelt es sich bei *Assignments* um konkrete MapReduce Aufträge. Infolgedessen setzt sich ein *Assignment*-Objekt zum einen aus dem *MapReduce Auftrag* und zum anderen aus dem dazugehörigen *Konfigurations*-Objekt zusammen. Da es sich bei den zwei Objekten tatsächlich um die entsprechenden Instanzen aus der *Hadoop Bibliothek* handelt, können sie ohne weitere Transformationen im MapReduce Framework ausgeführt werden. Die Reihenfolge wird dabei durch die *Assignment-Sequenz* festgelegt, welche nach Ausführung aller MapReduce

Aufträge den Zugriff auf Messwerte ermöglicht. Diese stellen die Grundlage für die durchgeführte Evaluation dar. Abschließend sei noch angemerkt, dass es sich bei der Abbildung 29 um eine vereinfachte Darstellung handelt, die nur den systematischen Ablauf der einzelnen Schritte veranschaulichen soll.

MapReduce API

Wie zuvor erläutert, müssen für die Ausführung von MapReduce Aufträgen Objekte aus der *Hadoop Bibliothek* und insbesondere aus dem MapReduce Framework erzeugt werden. Hierfür stellten die Hadoop Entwickler eine gut dokumentierte *MapReduce Programmierschnittstelle (API)* bereit.

Mit der Hadoop Version 0.20.0 wurde eine neue Version der MapReduce API veröffentlicht. Damit sollte vor allem eine solide Grundbasis für zukünftige Weiterentwicklungen geschaffen aber auch der allgemeine Umgang, also die Art und Weise wie Entwickler Hadoop Anwendungen implementieren können, vereinfacht werden. Die strukturellen Änderungen resultierten allerdings in einer Inkompatibilität zwischen der neuen und der alten API. Das hat zur Folge, dass innerhalb eines MapReduce Auftrages auf nur eine der beiden Schnittstellen zugegriffen werden darf. Im Rahmen dieser Arbeit hat es sich außerdem gezeigt, dass in der neuen MapReduce API einige spezielle Bibliotheken noch nicht vollständig portiert wurden[27]. Da allerdings in Zukunft die Unterstützung für die alte API aus den *Hadoop Distributionen* entfernt werden soll und die fehlenden Funktionalitäten der neuen API umgangen werden konnten, wurde für diese Arbeit ausschließlich die neue API herangezogen.

4.4 Triple-Loader

In diesem Abschnitt wird beschrieben, wie RDF-Graphen in den RDFPath-Store eingepflegt werden. Dieser Vorgang muss im Vorfeld etwaiger Anfragen einmalig durchgeführt werden und bildet folglich auch eine Grundvoraussetzung für die zu analysierenden RDF-Graphen. Die hierfür notwendigen Strukturen wurden in dem Begriff *Triple-Loader* zusammengefasst. Im Folgenden wird der konzeptuelle Ablauf eines solchen Prozesses in mehreren Schritten erläutert.

Ausgehend von einem RDF-Dokument, welches im Format RDF/Notation 3 (N3)[28] vorliegt, müssen die Triples im Vorfeld als eine zusammenhängende Da-

[27] Beispiele: `MultipleInputs, KeyValueTextInputFormat, MultipleOutputs, …`

[28] Eingeführt in Kapitel 2.1

tei in das Hadoop Distributed Filesystem kopiert worden sein. Wie in Abbildung 30 schematisch dargestellt, wird die Datei im ersten Schritt zeilenweise eingelesen. Daraufhin wird jedes Triple von einem *Parser* in seine drei Bestandteile Subjekt, Prädikat und Objekt aufgeteilt und wahlweise um Präfixe ergänzt. Im Rahmen dieser Arbeit wurden mehrere *Parser* implementiert, die an individuelle Charakteristika unterschiedlicher Datenquellen angepasst wurden. Dies war notwendig, um beim Einlesen der RDF-Dokumente bestimmte Kanten, die nicht benötigt wurden, auszufiltern. Auch wurden die Bezeichner auf unzulässige Sonderzeichen kontrolliert und, um die Übersichtlichkeit der in RDFPath spezifizierten Anfragen zu steigern, die Kantennamen gekürzt. Abschließend kann in diesem Schritt, wie in Abbildung 30 zu sehen ist, das im nachfolgenden Abschnitt 4.5 beschriebene *Dictionary Encoding* verwendet werden, um Subjekt, Prädikat und Objekt in eine kompaktere Darstellung zu überführen.

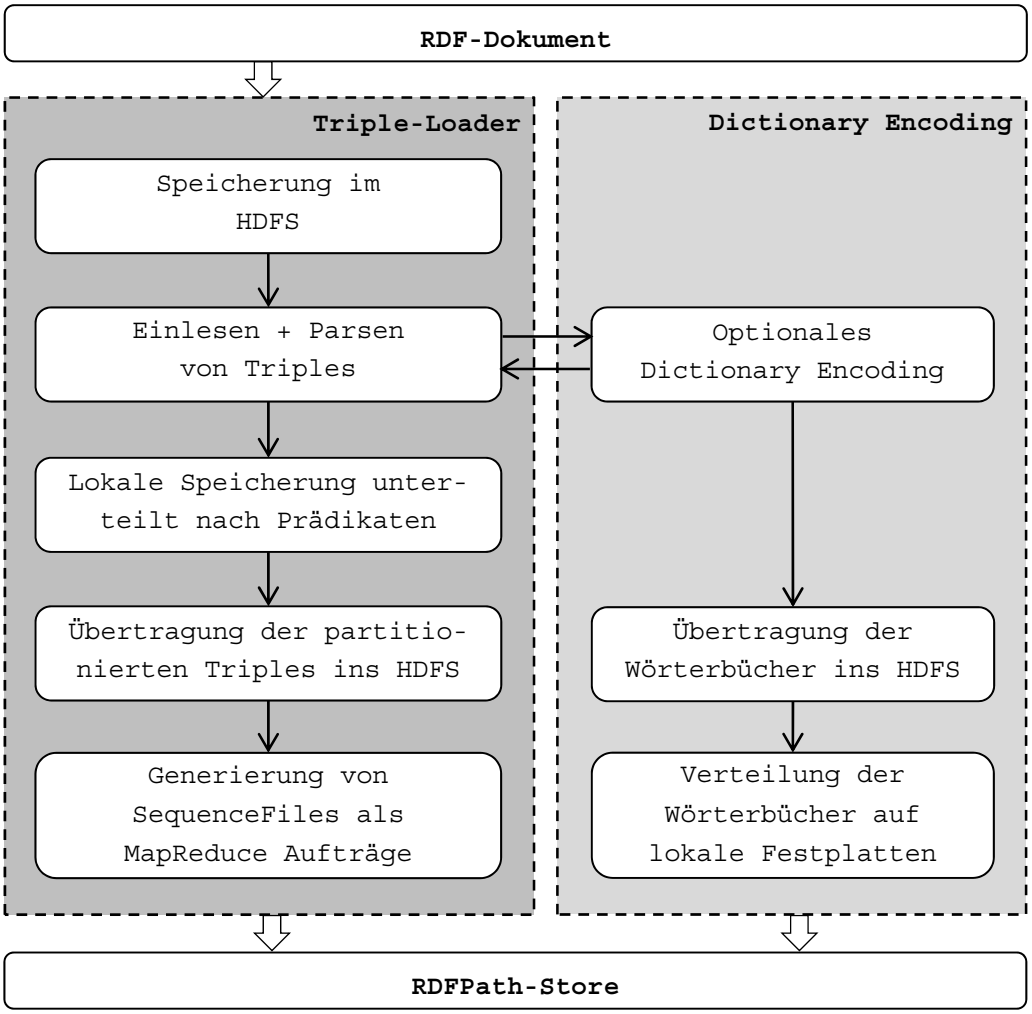


Abbildung 30: Schematischer Ablauf beim Einfügen von RDF-Dokumente

Im nächsten Schritt wird die vertikale Partitionierung bezüglich des Prädikates durchgeführt. Hierfür wird für jedes noch nie vorgekommene Prädikat eine lokale Datei angelegt und das *Tupel (Subjekt, Objekt)* darin abgespeichert. Das Prädikat muss hierbei nicht mitgespeichert werden, da es im Dateinamen aufgeführt wird. Dementsprechend wird jedes weitere Triple, welches ein bereits betrachtetes Prädikat aufweist, am Ende der entsprechenden Datei angehängt.

Nachdem das RDF-Dokument vollständig betrachtet wurde, werden der partitionierte Datenbestand sowie ggf. auch ein *Wörterbuch* in das HDFS verschoben. Im letzten Schritt werden in mehreren MapReduce-Phasen aus dem partitionierten Datenbestand die in Kapitel 4.1.2 spezifizierten `SequenceFiles` erzeugt, die als Eingabe für Reduce-Side-Joins fungieren können. Falls das Dictionary Encoding angewendet wurde, wird in dieser Phase auf allen Rechnern eine *lokale Kopie* der erzeugten Wörterbücher bereitgestellt, um möglichst kurze Zugriffszeiten zu gewährleisten.

4.5 Dictionary Encoding

Bei der letzten implementierten Komponente handelt es sich um eine Optimierungsstrategie, um die Größe von RDF-Dokumenten, die in den RDFPath-Store eingepflegt werden, zu reduzieren. Das *Dictionary Encoding* sollte die Datenmengen, die während der Auswertung einer Anfrage verarbeitet und über das Netzwerk transferiert werden, verringern. In Folgenden wird zunächst die Grundidee des Dictionary Encodings dargelegt. Abschließend werden die konkrete Implementierung und die verwendete Datenstruktur vorgestellt.

Das Dictionary Encoding ist eine Technik, die einzelne Buchstaben oder beliebige Zeichenfolgen, im Folgenden auch *Wörter* genannt, in eine andere Repräsentationsform abbildet. Eine solche *Abbildung* kann als eine eindeutige *Zuordnung* eines Wortes `A` zu einem Wort `B` verstanden werden, die beim erstmaligen Auftreten des Wortes `A` in einem *Wörterbuch* abgelegt wird. Demzufolge würde das Wörterbuch in diesem Fall folgenden *Eintrag* enthalten:

```
A -> B
```

Wenn das Dictionary Encoding auf eine größere Menge an Worten angewendet wird, kann es vorkommen, dass ein bereits betrachtetes Wort `A` erneut vorkommt. Da die Abbildung eindeutig sein soll, muss in solchen Fällen die zuvor verwendete Zuordnung im Wörterbuch nachgeschlagen und erneut angewendet werden. Ein weiterer *Wörterbucheintrag* für `A` ist nicht zulässig. Weiterhin muss beim Dictionary Encoding auch häufig die *Rücktransformation* des Wortes `B` in das ursprüngliche Wort `A` betrachtet werden. Während man bei der

Abbildung eines beliebigen Wortes `A` in eine andere Darstellungsform `B` von einer *Enkodierung* spricht, wird der Rückweg als *Dekodierung* bezeichnet. Für die Dekodierung eines Wortes ist ein weiteres Wörterbuch erforderlich, dass entsprechend dem fortlaufenden Beispiel folgenden Eintrag enthalten muss:

```
B -> A
```

Für die Speicherung von RDF Triples liegt der primäre Nutzen einer Dictionary Encoding Strategie in der Reduktion der Datengröße [Neumann, et al., 2008]. Lange Zeichenketten können häufig durch viel kürzere Repräsentanten ersetzt werden, sodass Daten in einem deutlich geringeren Umfang gelesen und verarbeitet werden müssen. Hierbei können, je nach Verwendungszweck, unterschiedliche Techniken wie beispielsweise *Hash-Funktionen*, *B-Bäume* oder *direktes Mapping* zum Einsatz kommen [Eickler, et al., 1995].

Im Rahmen dieser Arbeit wird das Dictionary Encoding eingesetzt, um die Bezeichner von Subjekten, Prädikaten und Objekten auf *inkrementierende Zahlenwerte* abzubilden. Hierbei werden zwei separate Wörterbücher, jeweils eines für die Enkodierung und eines für die Dekodierung verwendet. Da es üblicherweise deutlich weniger unterschiedliche Prädikate als Subjekte und Objekte gibt, hat es sich angeboten, zwei zusätzliche Wörterbücher für die Enkodierung und Dekodierung von Prädikaten gesondert zu verwalten.

Die implementierte Dictionary Encoding Strategie ordnet nun jedem neuen Subjekt, Prädikat oder Objekt, den der Parser im Triple-Loader einliest, einen noch nicht vorgekommenen Zahlenwert zu. Dies wird gewährleistet, indem zunächst im Wörterbuch nachgeschlagen wird, ob der aktuelle Bezeichner bereits zuvor betrachtet wurde. Falls nicht, wird der zuletzt vergebene Zahlenwert um eins inkrementiert. Hierbei kommen zwei unterschiedliche Zahlenwerte zum Einsatz, die auch unabhängig voneinander inkrementiert werden. Der erste Zahlenwert wird bei Subjekten sowie Objekten und der zweite bei Prädikaten angewendet. Dies ist zulässig, da die Prädikate zum einen gemäß der Syntax von RDFPath immer aus ihrem Kontext heraus von den übrigen Objekten unterschieden werden können und zum anderen, weil sie in einer separaten Datenstruktur abgelegt werden.

Durch jeden neu zugordneten Zahlenwert entstehen zwei neue Wörterbucheinträge, die in zwei separaten Wörterbüchern abgelegt werden. Dieses Prinzip wird in Abbildung 31 grafisch veranschaulicht. Wie in der beispielhaften Darstellung zu sehen ist, werden insgesamt vier unterschiedliche Wörterbücher benötigt. Jedes dieser Wörterbücher wurde mittels einer gesonderten *Berkeley Database* realisiert, die in Kapitel 2.3 eingeführt wurde. Da die Berkeley DB dafür gedacht ist, Schlüssel-/Wertpaare aufzunehmen, und die Zuordnung eines

Bezeichners zu einem Zahlenwert und der umgekehrte Fall genau diesem Schema entsprechen, konnte sie ohne größere Umwege direkt eingebunden werden.

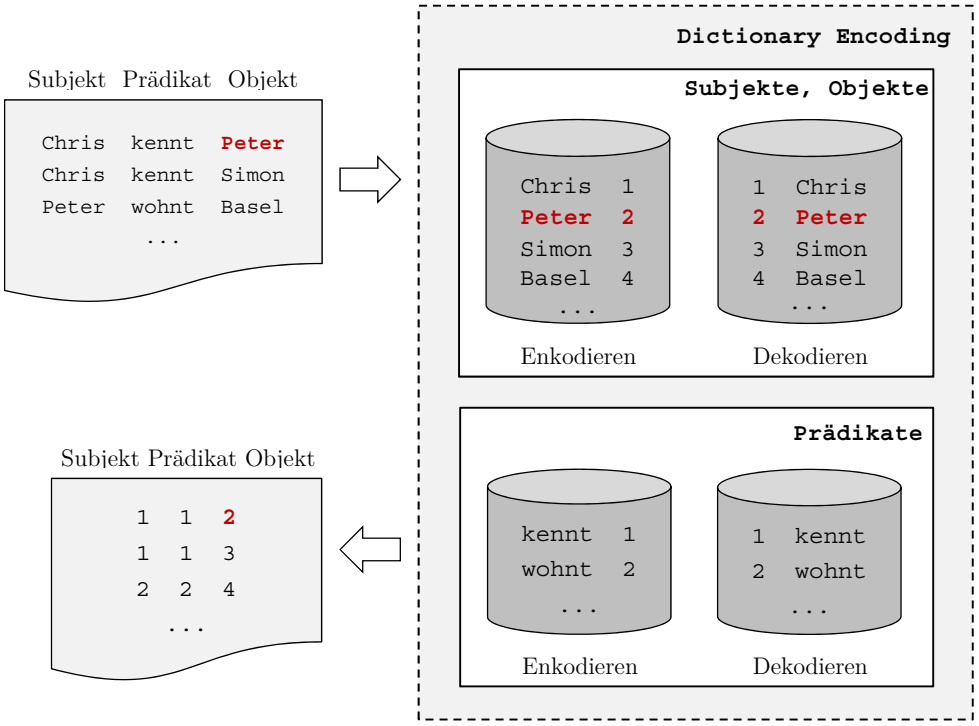


Abbildung 31: Beispiel - Dictionary Encoding

Bei der Berkeley DB wurden als Datenstruktur *B-Bäume* gewählt. Da ein gleichzeitiger Mehrfachzugriff ausgeschlossen werden konnte, wurden darüber hinaus die *Transaktions-* und *Sperrmechanismen* deaktiviert. Ebenso wurde die Möglichkeit, Änderungen an bereits eingefügten Datensätzen durchzuführen, ausgeschaltet. Da bei der späteren Auswertung von Anfragen lediglich lesend auf die Datenbanken zugegriffen wird, kommt in diesem Zusammenhang der *„Read-Only“-Modus* zum Einsatz. Diese Konfiguration wurde im Hinblick auf möglichst kurze Zugriffszeiten optimiert.

Mit diesem Abschnitt wurde nun die letzte Komponente des entwickelten Systems vorgestellt. Um die anfänglich festgelegten Fragestellungen dieser Arbeit beantworten zu können, musste eine Evaluation der Implementierung durchgeführt werden, die im nächsten Kapitel vorgestellt wird.

5 Evaluation

In diesem Kapitel werden die Eigenschaften der zuvor vorgestellten Implementierung auf unterschiedlichen Datenbeständen getestet. Das Ziel dieser *Evaluation* besteht dabei darin herauszufinden, ob Analysen von großen RDF-Graphen mit den ausgewählten Techniken und entwickelten Strategien sinnvoll durchgeführt werden können und wie sich die Auswertungszeiten bei steigenden Datengrößen verhalten.

Dieses Kapitel gliedert sich in vier Abschnitte. Zunächst werden in den Grundlagen die Voraussetzungen, unter denen die Evaluation durchgeführt wurde, aufgezeigt. In den beiden darauffolgenden Abschnitten werden zwei Evaluation, die auf unterschiedlichen Datenbeständen durchführt wurden, vorgestellt. Abschließend werden im letzten Abschnitt die Ergebnisse zusammengefasst und die noch offen gebliebenen Fragestellungen besprochen.

5.1 Grundlagen

Im Vorfeld der Evaluation müssen zunächst die Bedingungen, unter denen die Evaluation durchgeführt wurde, beschrieben werden. Hierzu zählen die Hardwareausstattung, die installierte Software, die gewählten Konfigurationseinstellungen und natürlich auch die Datenbestände, auf denen die Evaluation durchgeführt wurde. Neben den eigentlichen Datenbeständen werden noch einige alternative Datenquellen und der Entscheidungs- sowie Beschaffungsprozess kurz beleuchtet. Abschließend werden noch Rahmenbedingungen aufgestellt, nach denen die Anfragen für die darauffolgende Evaluation konzipiert worden sind.

5.1.1 Hard- und Software

Hardwareseitig wurde ein Rechencluster, bestehend aus zehn Dell PowerEdge R200 Servern, eingesetzt. Jeder Server verfügte über einen Dual Core Xeon E3120 3,16 GHz Prozessor, 4 GB DDR2 SDRAM 800 MHz, eine 1 TB SATA Festplatte mit 7.200 U/min sowie einem Dual Port Gigabit Ethernet Adapter. Die zehn Server wurden über einen 3Com Baseline Switch 2824 in einem Gigabit-Netzwerk miteinander verbunden.

Softwareseitig wurde auf dem Rechencluster Ubuntu 9.10 (x86_64) als Betriebssystem eingerichtet und Java in der Version 1.6.0_15 verwendet. Zudem

wurde auf allen Servern *Cloudera's Distribution for Hadoop 3* (CDH3)[29] installiert. Hierbei handelt es sich um ein gut dokumentiertes und leicht zu installierendes Paket, das die Hadoop Version 0.20.2 um einige Fehlerbereinigungen und zusätzliche Funktionen ergänzt. Es wird vom Hadoop-Dienstleister *Cloudera* in regelmäßigen Abständen aktualisiert und soll Unternehmen wie Entwicklern den Einstieg in Hadoop erleichtern.

In Anlehnung an eine hierarchische *Master-Slave Architektur* fungiert einer der zehn Server als *Master*. Er koordiniert die Verteilung sowie Verwaltung von MapReduce Aufträgen (`JobTracker`) und übernimmt die Speicherung aller Metadaten, der im HDFS abgelegten Dateien (`NameNode`). Die restlichen neun Server (*Slaves*) sind für die Speicherung der Daten (`DataNodes`) und die tatsächliche Ausführung der MapReduce Aufträge (`TaskTracker`) zuständig. Dabei läuft auf jedem Slave üblicherweise sowohl eine `DataNode` als auch eine `TaskTracker`-Instanz. An der voreingestellten Standard-Konfiguration von Cloudera wurden nur wenige Änderungen vorgenommen. Der *HDFS Replikationsfaktor* wurde auf drei belassen, sodass von anfänglichen 8 Terabyte an Festplattenspeicher effektiv nur ca. 2,5 Terabyte für Anfragen genutzt werden konnte. Allerdings wurden zwei Einstellungen bezüglich des Arbeitsspeichers angepasst. So wurde die *Heapsize* des *Hadoop Deamons* auf zwei Gigabyte erhöht und die *Heapsize* der sogenannten *Child Java Virtual Machines (JVM)* auf ein Gigabyte gesetzt. Weiterhin wurde noch die maximale Anzahl an *parallelen Reducer-Instanzen* auf neun eingeschränkt, sodass auf jedem Slave-Server nur ein einziger Reducer gestartet wurde. Es ist davon auszugehen, dass weitergehende Anpassungen der zahlreichen Konfigurationsparameter eine Verbesserung der gemessen Ausführungszeiten ermöglichen sollten.

5.1.2 Daten

Neben der Implementierung der Pfadanfragesprache und aller zugehörigen Komponenten war - insbesondere im Hinblick auf die nachfolgende Evaluation - die Beschaffung von RDF-Daten ein weiterer wichtiger Aspekt, den es zu berücksichtigen galt. Dabei war die Wahl der richtigen Datenquelle sowohl für kleine Tests als auch für die durchgeführte Evaluation von zentraler Bedeutung. So sollten die Datenbestände, insbesondere im Hinblick auf das Ziel dieser Arbeit, RDF-Graphen unter Verwendung einer Pfadanfragesprache zu analysieren, mehreren Anforderungen gerecht werden. Zunächst sollten sie in einem RDF Format vorliegen oder sich zumindest mittels einer Konvertierung

[29] http://www.cloudera.com/hadoop/

als solche darstellen lassen. Weiterhin war es wichtig, dass die Daten einen verbundenen Graphen repräsentieren, auf dem auch längere Pfade betrachtet werden konnten. Die zahlreichen Ausdrucksmöglichkeiten von RDFPath, zu denen beispielsweise Unteranfragen, Filter und die Suche nach kürzesten Pfaden zählen, sollten in unterschiedlichen Kombinationsmöglichkeiten zweckmäßig eingesetzt werden können. Darüber hinaus wäre es noch wünschenswert, wenn die Graphen gewisse *Small World Charakteristika* aufweisen würden, die man mit entsprechenden Anfragen analysieren könnte [Ziegler, 2010]. Die weiteren Anforderungen beziehen sich auf den Umfang der Datenbestände. Dieser sollte mindestens 10 Gigabyte umfassen, mehrere Millionen Triples aufweisen und innerhalb von nur wenigen Wochen verfügbar sein. Zudem war es erforderlich, die Daten in unterschiedliche Größen unterteilen zu können, um zu erfahren, wie sich ein zunehmender Datenumfang auf die Auswertungszeit einer Anfrage auswirkt.

Im Rahmen dieser Arbeit wurden mehrere unterschiedliche Datenquellen betrachtet und entsprechend der zuvor vorgestellten Anforderungen eingeschätzt. Tabelle 5 zeigt hierzu eine kurze Übersicht. Wie man aus der Tabelle ablesen kann, standen schlussendlich nur noch vier Datenquellen zur Auswahl: *Flickr, Last.fm,* der *Jung-Generator* und der *SP²Bench Generator*. Da die ersten drei Datenquellen zusätzlichen Programmieraufwand und einer weiteren Einarbeitungsphase bedurften, konnte nur schwierig abgeschätzt werden, wann die entsprechenden Daten verfügbar wären. Folglich fiel die Entscheidung zunächst zu Gunsten des SP²Bench Generators aus. Die hierdurch erzeugten *DBLP Daten* waren in der erforderlichen Größe sofort verfügbar und da Publikationen per se vernetzt sind, konnten sie auch als ein zusammenhängender Graph verstanden werden.

Zwar konnte man auf den generierten *DBLP Daten* die zahlreichen Ausdrucksmöglichkeiten von RDFPath ausprobieren, komplexere Anfragen erreichten allerdings schnell einen recht künstlichen Charakter und waren somit schwer zu interpretieren. Dies begründete die Betrachtung einer weiteren Datenquelle. Da die erste synthetisch generiert wurde, sollte die zweite Datenquelle reale Daten repräsentieren. Entsprechend der Tabelle 5 blieb folglich nur noch der Bilderdienst *flickr* und die Musikplattform *Last.fm* übrig. Bei beiden Kandidaten handelt es sich um soziale Netzwerke, die den Zugriff über eine im Internet verfügbare *Programmierschnittstelle* gestatteten. Die Datenbestände von Last.fm suggerierten jedoch mehr Möglichkeiten, einfach zu interpretierende Pfadanfragen auszudrücken. Zudem setzte das Unternehmen bereits selbst Hadoop‘s MapReduce Framework ein. Folglich wurde als zweite Datenquelle

Last.fm ausgewählt. Als nächstes werden die Daten aus dem SP²Bench Generator und der Beschaffungsprozess der Last.fm Daten beschrieben.

Datenquelle	Beschreibung	Einschätzung
IMDb (Amazon.com)	Datenbank für Filme, Serien, Videoproduktionen sowie mitwirkenden Personen ~150.000 Filme, ~2.100.000 Einträge Größe: ca. 7 GB (komprimiert)	Sinnvolle Anfragen, Datenformat aufwändig zu konvertieren
freeDB	Datenbank mit CD Informationen ~3 Mio. CDs Größe: ca. 5 GB	Kein zusammenhängender Graph
Audioscrobbler (2005)[30]	Hör-Profile von realen Menschen ~ 150.000 Einträge Größe: ca. 450 MB	Zu kleiner Umfang
DBLP	Datenbank für Publikationen ~ 1.5 Mio. Einträge	Small World Charakteristika, zu kleiner Umfang
flickr (Yahoo)	Struktur eines sozialen Netzwerkes, Bildern, Profile und Freundschaften Zugriff über Online API	Sinnvolle Anfragen, Small World Charakteristika, Webcrawler notwendig
Last.fm	Struktur eines sozialen Netzwerkes, Musiktitel, Interpreten, Profile, Freundschaften, Hör-Profile	Sinnvolle Anfragen, Small World Charakteristika, Webcrawler notwendig
JUNG	Java Framework für Graphen, Generator unterstützt Kleinberg's Modell	Variable Größe, synthetische Small World Charakteristika, keine Erfahrungswerte
SP²Bench	Generator für DBLP Daten	Variable Größe, verfügbar + nutzbar

Tabelle 5: Übersicht an potenziellen Datenquellen für die Evaluation

[30] Der Vorläufer von Last.fm

SP²Bench Daten

Der *SP²Bench Generator* wurde im Rahmen des *SPARQL Performance Benchmarks* (SP²Bench) konzipiert, einem Benchmark, der mit seinen Anfragen die speziellen Besonderheiten von SPARQL berücksichtigt [Schmidt, et al., 2009]. Mit dem Generator lassen sich nun beliebig große RDF-Dokumente erzeugen, die *bibliografische Informationen* über synthetische Publikationen enthalten. Als Grundlage hierfür werden Wahrscheinlichkeitsverteilungen sowie weitere abgeleitete Eigenschaften einer echten *DBLP-Bibliothek* herangezogen. In Abbildung 32 ist das generierte RDF-Dokument als Graph dargestellt.

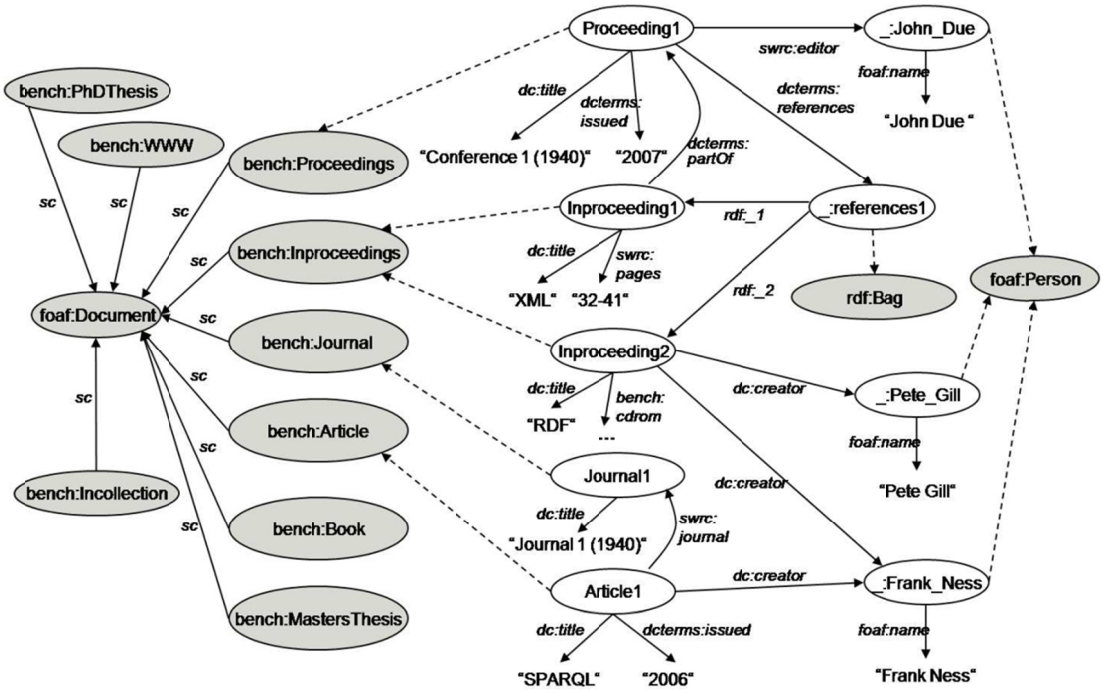


Abbildung 32: RDF-Graph des SP²Bench Generators [Schmidt, et al., 2009]

Allerdings erwies sich hier das Ausdrücken von längeren Pfaden noch als recht umständlich. Abhilfe schaffte die Einführung sogenannter Rückwärtskanten. So wurde beispielsweise zu einem Triple `(Article1, creator, Frank)` ein weiteres Triple `(Frank, r_creator, Article1)` erzeugt[31]. Somit hatte man die Möglichkeit, ausgehend von einem Autor mit einem weiteren Lokationsschritt alle seine Publikationen zu betrachten, was ohne die Rückwärtskanten in RDFPath nicht möglich gewesen wäre.

[31] Dies kann im Triple-Loader (Kapitel 4.4) optional aktiviert werden.

In Tabelle 6 werden die unterschiedlichen Größen der generierten RDF-Dateien aufgeführt, die als Grundlage für die nachfolgende Evaluation verwendet worden sind. Hierbei ist anzumerken, dass die kleineren Datenbestände in den jeweils Größeren enthalten sind.

# Triples:	**0,01 Mio**	**0,05 Mio**	**0,25 Mio**	**1 Mio**	**5 Mio**	**25 Mio**
Dateigröße:	1 MB	5 MB	26 MB	106 MB	536 MB	2,7 GB
# Triples:	**50 Mio**	**100 Mio**	**200 Mio**	**400 Mio**	**800 Mio**	**1600 Mio**
Dateigröße:	5,3 GB	10,7 GB	21,2 GB	42,1 GB	83,8 GB	167,3 GB

Tabelle 6: Vom SP²Bench-Generator erzeugte Triples

Last.fm Daten

Bei *Last.fm* handelt es sich um einen Empfehlungsservice für Musik. Die Teilnehmer müssen sich hierfür zunächst registrieren und können danach unter Verwendung einer zusätzlichen Software oder des integrierten Internetradios Informationen über abgespielte Musiktitel und somit auch den eigenen Musikgeschmack an Last.fm übermitteln. Die übermittelten Daten werden mit denen anderer Teilnehmer verglichen, um beispielsweise Empfehlungen für weitere Titel bereitzustellen oder Menschen kennenzulernen, die einen ähnlichen Musikgeschmack haben. Auf dieser Grundlage ist ein *soziales Netzwerk* entstanden, das – ähnlich wie Facebook – allen Teilnehmern eine Internetplattform mit Freundeslisten und Kommunikationsmöglichkeiten offeriert. Dieses soziale Netzwerk bildet den Ausgangspunkt für die benötigen RDF-Daten, denn die Freundschaftsbeziehungen der Teilnehmer und die Zusammenstellungen von Lieblingsartisten oder Lieblingstitel lassen sich in einem zusammenhängenden RDF-Graphen abbilden, der wiederrum mit RDFPath analysiert werden kann. Der Zugriff auf diese Informationen erfolgt über eine im Internet zugreifbare *Programmierschnittstelle*, die im Rahmen einer offenen Lizenzpolitik[32] in Anspruch genommen werden kann.

Für das Sammeln entsprechend großer Datenmengen muss ein *Webcrawler* implementiert werden, der gewünschte Informationen automatisiert abruft und lokal abspeichert. Zu diesem Zweck werden auf Last.fm fertige *Bibliotheken*[33] angeboten. Sie sollen den Zugriff auf die Daten noch weiter erleichtern und

[32] http://www.last.fm/api/tos, (4. License to use the Last.fm Data)

[33] Externe Projekte ohne Last.fm Unterstützung

stehen für C++, Java, PHP und andere Programmiersprachen zur Verfügung. Die Implementierung, die dieser Arbeit zu Grunde liegt, beruht auf einer Bibliothek für die Skriptsprache PHP[34]. Der implementierte Webcrawler stellt folglich ein PHP-Skript dar, das auf einem Webserver ausgeführt werden kann. Für das Sammeln der Daten werden dabei zunächst diejenigen Informationen spezifiziert, die über die Last.fm Programmierschnittstelle abgerufen werden sollen. Sie können im Kontext von RDF-Graphen auch als Kanten verstanden werden, die einen Zusammenhang zwischen einem Subjekt und einem Objekt beschreiben. Anschließend wird, ausgehend von einem Startknoten, der beispielsweise ein Teilnehmer von Last.fm oder ein Artist sein kann, eine *Breitensuche* durchgeführt. Die Suche betrachtet alle zuvor ausgewählten Kanten und speichert die abgerufenen Daten als Triple der Form `(Subjekt, Prädikat, Objekt)` in einer *zusammenhängenden Datei zeilenbasiert* ab. Nachdem die gewünschte Anzahl an Triples gesammelt wurde, wird die Suche abgebrochen. Die erzeugte Datei kann abschließend in das HDFS kopiert und mit dem Triple-Loader eingelesen werden.

Insgesamt wurden 225 Millionen Triple gesammelt, die eine Größe von 13 Gigabyte aufwiesen. Abbildung 33 zeigt den betrachteten Graphen in einer vereinfachten Darstellung[35].

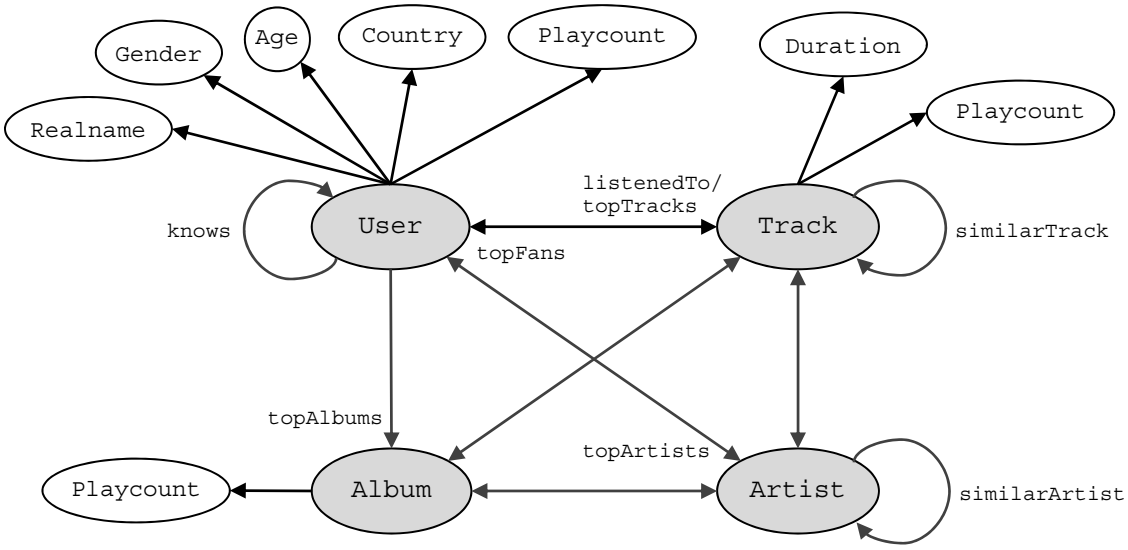


Abbildung 33: Last.fm Daten als Graph

[34] http://www.php.net/

[35] Die fehlenden Kanten wurden nach den Knoten benannt, die sie verbinden.
Beispiel: Die Kante von „`User`“ nach „`Age`“ heißt „`userAge`“.

In der Tabelle 7 werden die unterschiedlichen Größen der gesammelten RDF-Dateien aufgeführt, die als Grundlage für die nachfolgende Evaluation verwendet worden sind. Auch hier gilt, dass die kleineren Datenbestände in den jeweils Größeren enthalten sind.

# Triple:	**1 Mio.**	**2 Mio.**	**4 Mio.**	**8 Mio.**	**16 Mio.**
Dateigröße:	57 MB	114 MB	227 MB	474 MB	928 MB
# Triple:	**25 Mio.**	**50 Mio.**	**100 M.**	**200 Mio.**	**225 Mio.**
Dateigröße:	1,5 GB	2,8 GB	5,8 GB	11,4 GB	12,8 GB

Tabelle 7: Vom Musikdienst Last.fm gesammelte Triples

5.1.3 Rahmenbedingungen für Benchmark Anfragen

Der letzte Punkt, der die Grundlagen für die Evaluation vervollständigt, betrifft die *Kriterien*, nach denen die Anfragen konzipiert werden sollten sowie die Erzeugung von *Messwerten*. In diesem Zusammenhang musste zunächst festgelegt werden, welche Aspekte dieser Arbeit mit der Evaluation abgedeckt werden sollten. Dabei war es von besonderer Wichtigkeit, auch die Charakteristika der Daten einzubeziehen und die Anfragen entsprechend der Konnektivität und des Verzweigungsgrades anzupassen.

Die *Benchmark-Anfragen* wurden unter Berücksichtigung der nachfolgenden Kriterien konzipiert.

(1) Das *Skalierungsverhalten* der ganzen Implementierung und insbesondere der Reduce-Side-Joins sowie die Suche nach dem kürzesten Pfaden soll untersucht werden.

(2) Die Eigenschaften des Dictionary Encodings sollen analysiert werden, indem eine im Vergleich zur Eingabe-/Ausgabemenge relativ hohe Anzahl an Zwischenergebnissen generiert wird.

(3) Die *Kosten* für das Dictionary Encodings sollen aufgezeigt werden, indem eine zur Eingabe-/Ausgabemenge relativ niedrige Anzahl an Zwischenergebnissen generiert wird, während gleichzeitig sehr viele Wörterbuchzugriffe erforderlich werden.

(4) Die Ausdrucksmöglichkeiten von RDFPath sollen beispielhaft dargelegt werden.

(5) Es sollen Vergleichsmöglichkeiten für Teilaspekte der Implementierung zu anderen Sprachen wie PIG oder SPARQL untersucht werden.

Ferner war für die Erfassung des *Leistungsverhaltens* einzelner Anfragen die Generierung und Speicherung unterschiedlicher *Messwerte* erforderlich. Hierfür wurde ein *Messsystem* entwickelt, das fest in die Implementierung eingebettet ist und auf sämtliche Konfigurationsparameter sowie instanziierte Objekte direkt zugreifen kann. Nachfolgend werden einige wesentliche Messwerte, die gespeichert werden, aufgeführt.

(1) **Ausführungszeit:** Dies stellt die Zeit dar, die für die Auswertung einer Anfrage im MapReduce-Framework benötigt worden ist. Der Startpunkt beschreibt hierbei die letzte Anweisung bevor die Ausführung des ersten MapReduce Jobs in Hadoop gestartet wird. Der Endpunkt beschreibt die erste Anweisung nach Ausführung aller MapReduce Jobs, die Bestandteil der Anfrage gewesen sind. Eine solche Zeitmessung wird häufig auch als *Walltime* bezeichnet.

(2) **Übertragene Daten:** Während der Auswertung müssen zwischen der Map- und der Reduce-Phase (also in der sogenannten Shuffle-Phase) Daten durch das Netzwerk, von einem Server zum anderen, übertragen werden. Dieser Messwert beschreibt nun die Menge der Daten, die von allen MapReduce Jobs im Netzwerk übertragen werden. Hierbei werden die Datenübertragungen, die das HDFS zum Erzeugen des gewünschten Replikationsfaktors benötigt, nicht mitgerechnet.

(3) **Geschriebene HDFS Daten:** Dieser Wert beschreibt die Menge an Daten, die von allen MapReduce Jobs in das HDFS geschrieben werden.

(4) **Geschriebene File Daten:** Dieser Wert beschreibt die Menge an Daten, die von allen MapReduce Jobs nicht in das HDFS, sondern auf die lokale Festplatte geschrieben werden.

(5) **Ergebnisse:** Zu jeder Anfrage werden die Anzahl an Ergebnissen sowie eine zufällige Auswahl an fünf Einträgen aus der gesamten Ergebnismenge abgespeichert.

5.2 DBLP Evaluation

In diesem Abschnitt wird die Auswertung von mehreren RDFPath Anfragen, die auf den zuvor vorgestellten Daten des *SP²Bench Generators* ausgeführt wurden, vorgestellt. Um Aussagen über die *Skalierbarkeit* treffen zu können, wurden die Anfragen auf unterschiedlichen Datengrößen ausgeführt, die bei Bedarf in der Tabelle 6 nachzulesen sind. Bevor nachfolgend einige ausgewählte Benchmarks vorgestellt werden sei angemerkt, dass die betrachteten Anfragen, insbesondere auch durch die Verwendung von Rückwärtskanten, oftmals einen recht künstlichen Charakter aufweisen und somit nicht immer leicht zu interpretieren sind.

Dictionary Encoding

Der erste Teil der Evaluation entspricht keiner ausführbaren Anfrage, sondern beschreibt einige Ergebnisse der implementierten Dictionary Encoding Strategie. Die zuvor vorgestellten DBLP Daten werden zum einen in ihrer ursprünglichen Textform betrachtet und zum anderen in einer enkodierten Fassung. In Abbildung 34 werden die resultierenden Dateigrößen einander gegenübergestellt. Hierbei wird der anfänglich genierte Datenbestand als „**Quelle**“ bezeichnet. Nach dem Einpflegen in den RDFPath-Store wird zwischen der *ursprünglichen Repräsentationsform* „**Text**“ und der *enkodierten Repräsentationsform* „**Enkodierung**“ unterschieden. Die Rückwärtskanten, die den Datenbestand annähernd verdoppeln, wurden für diese Abbildung außen vor gelassen. Außerdem wurden aus zeitlichen Gründen nur die Datenbestände bis 100 Millionen Triples enkodiert. Darüber hinaus ist nur die Textvariante verfügbar.

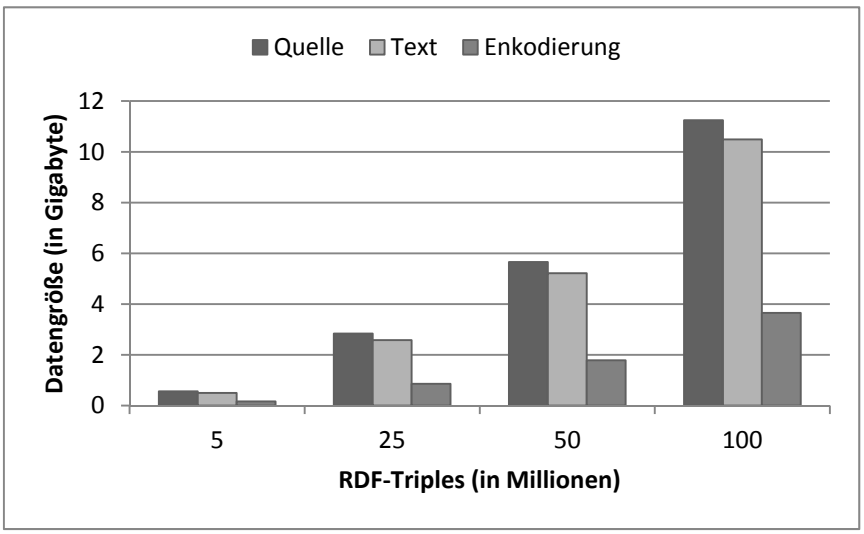


Abbildung 34: Dictionary Encoding und der Speicherplatzbedarf

In der Abbildung ist zu sehen, dass sich die Überführung der generierten Datenbestände in den RDFPath-Store bereits ohne Dictionary Encoding positiv auf die Datengröße auswirkt. Dies ist allerdings größtenteils auf einige ausgefilterte und gekürzte Prädikate zurückzuführen (siehe Kapitel 4.4). Weiterhin ist zu sehen, dass die enkodierte Repräsentationsform der Datenbestände erwartungsgemäß deutlich weniger Speicherplatz benötigt als die ursprüngliche Textform. In Zahlen ausgedrückt wird durch das Dictionary Encoding der Speicherplatzbedarf im Durchschnitt um ca. 65% gesenkt, wobei sich die Werte im Bereich von 63 bis 68% bewegen. Hierzu muss allerdings angemerkt werden, dass bei der Enkodierung der Triples natürlich auch noch Wörterbücher erzeugt wurden, die nochmal zusätzlichen Speicherplatz benötigen. Dieser ist in der Abbildung 34 nicht aufgeführt. Weitere Eigenschaften des Dictionary Encodings werden bei den entsprechenden Anfragen besprochen.

Anfrage 1

```
bench:Inproceedings :: r_type
                > creator[type=equals(foaf:Person)]
                > r_creator[type=equals(bench:Article)].
```

Abbildung 35: RDFPath-Anfrage 1

Bei dieser Anfrage werden durch die Verwendung der Rückwärtskante „`r_type`“ zunächst alle „`Inproceedings`“ lokalisiert. Anschließend werden zu jedem „`Inproceeding`“ die Autoren betrachtet, wobei mit einer Unteranfrage sichergestellt wird, dass alle Autoren den Typ „`foaf:Person`“ aufweisen. Ausgehend von den hierdurch bestimmten Autoren werden im letzten Lokationsschritt deren Publikationen bestimmt. Zudem werden durch eine Unteranfrage diejenigen Publikationen herausgefiltert, die nicht den Typ „`Article`“ aufweisen. Ein um seine Präfixe gekürzter Pfad, der bei der Auswertung dieser Anfrage generiert wurde, sah dabei wie folgt aus:

```
bench:Inproceedings (r_type) Inproceeding3 (creator) Paul_Erdoes
                        (r_creator) Article8
```

Diese Anfrage wurde auf der ursprünglichen Textvariante der Datenbestände und auf der enkodierten Variante ausgeführt. In der nachfolgenden Abbildung 36 sind nun die Ausführungszeiten bei einer steigenden Anzahl an RDF-Triples zu sehen.

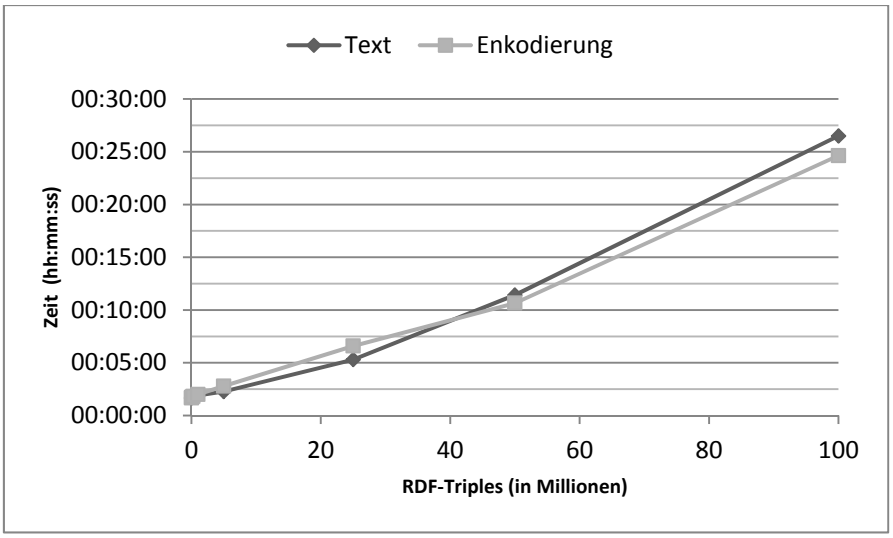


Abbildung 36: Diagramm zur Anfrage 1

Aus der Grafik lässt sich ableiten, dass die Ausführungszeiten der Anfrage, sowohl auf der Textvariante als auch auf den enkodierten Daten, linear in Abhängigkeit der *Eingabegröße (RDF-Triples)* zu steigen scheinen. Eine eindeutige Aussage lässt sich hier aufgrund der wenigen Messpunkte nur schwer treffen. Weiterhin lässt sich feststellen, dass die enkodierten Daten ab 40 Millionen RDF-Triples eine marginale Verbesserung der Ausführungszeit bewirken. Unter 40 Millionen Triples scheinen bei dieser Anfrage die nicht enkodierten Daten sogar vorteilhaft zu sein. Dieser Effekt wird bei der Last.fm Evaluation genauer beleuchtet.

Anfrage 2

```
bench:Article :: r_type > pages.
```

Abbildung 37: RDFPath-Anfrage 2

Bei dieser Anfrage wird zu jedem Artikel die Anzahl an Seiten betrachtet. Für die Auswertung dieser Anfrage wird nur ein Reduce-Side-Join benötigt, wobei ein beispielhafter Pfad, der durch diese Anfrage generiert wird, in gekürzter Form wie folgt aussieht:

```
bench:Article (r_type) Article5 (pages) 28
```

Die Intention hinter dieser Anfrage war es zum einen, die Grundkomponente der Implementierung, den Reduce-Side-Join, ohne zusätzlichen Konstrukte wie Filter oder Unteranfragen darzustellen. Zum anderen sollte mit dieser Anfrage eine Vergleichsmöglichkeit zu angrenzenden Implementierungen geschaffen werden. Da parallel zu dieser Arbeit eine Überführung von SPARQL nach

PigLatin, *PigSPARQL*[36], [Schätzle, 2010] implementiert wurde, bot sich in diesem Zusammenhang ein direkter Vergleich zwischen PigSPARQL und RDFPath an, der in Zusammenarbeit mit Alexander Schätzle durchgeführt wurde. Hier galt es allerdings zu beachten, dass beide Sprachen und demzufolge auch die zugrunde liegenden Implementierungen, für unterschiedliche Anwendungsbereiche konzipiert wurden. Ein entsprechender Vergleich ist daher eher als ein ungefährer Indikator zu verstehen, der nur bestimmte Teilaspekte wiedergeben kann. Nichtsdestotrotz war es interessant zu erfahren, wie zwei unterschiedliche Anfragen auf vergleichbaren Datensätzen eine ebenso vergleichbare Aufgabe lösen. Glücklicherweise konnten hierfür sogar SP²Bench-Anfragen herangezogen werden. So berechnet die hier zugrunde liegende Anfrage die gleichen Ergebnisse wie die Q3a aus [Schmidt, et al., 2009]. Sie ist der Vollständigkeit halber in der Abbildung 38 aufgeführt.

```
SELECT ?article
    WHERE {
      ?article rdf:type bench:Article .
      ?article ?property ?value
      FILTER (?property = swrc:pages)
    }
```

Abbildung 38: SP²Bench-Anfrage (Q3a) aus [Schmidt, et al., 2009]

Für beide Ausführungen wurden vertikal partitionierte Daten verwendet, die aus derselben Quelldatei stammen. Weiterhin stimmte auch die Anzahl an Ergebnissen überein. In Abbildung 39 sind die Ausführungszeiten für unterschiedliche Eingabegrößen zu sehen.

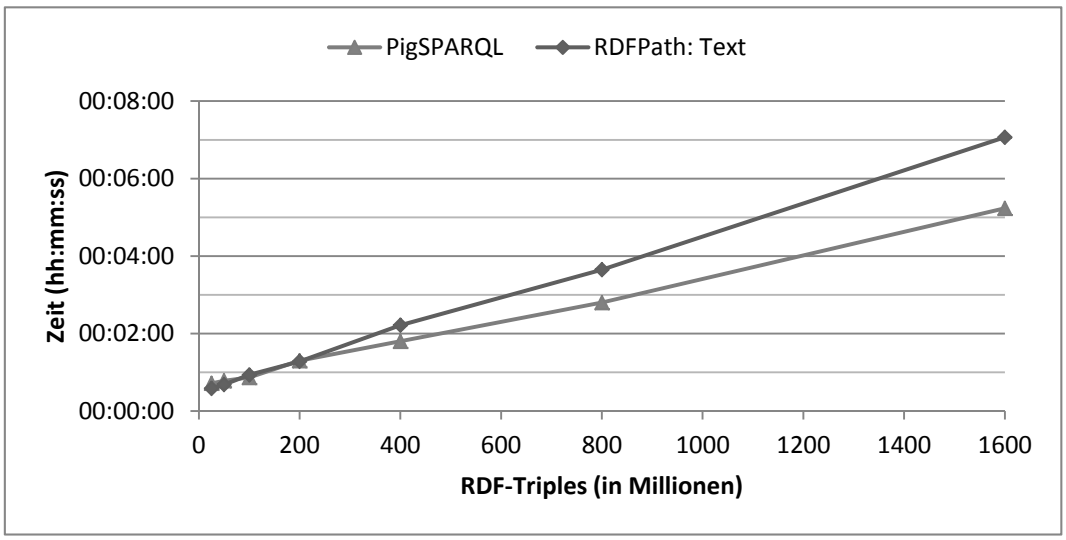


Abbildung 39: Diagramm zur Anfrage 2

[36] Wird ebenfalls im MapReduce-Framework ausgeführt (PIG: siehe Kapitel 2.2.1)

Bei beiden Implementierungen steigt die Ausführungszeit linear mit der Anzahl der betrachteten RDF-Triples an. Weiterhin lässt sich festhalten, dass die gemessenen Zeiten keine gravierenden Unterschiede aufweisen. Zwar ist PigSPARQL ab 200 Millionen Triples durchgehend schneller, jedoch darf hier nicht außer Acht gelassen werden, dass bei RDFPath beispielsweise ganze Pfade in die Ergebnismenge aufgenommen werden und die partitionierten Daten nicht als reiner Text sondern als SequenceFiles vorliegen.

Über diese Anfrage hinaus wurden noch weitere Vergleiche zwischen PigSPARQL und RDFPath angestrebt. Von den 17 SP²Bench Anfragen aus [Schmidt, et al., 2009] ließ sich allerdings nur noch die Q1 mit RDFPath annährend vergleichbar ausdrücken, sodass zumindest die gleiche Anzahl an Ergebnissen erzielt wurde. Auch hier wiesen beide Implementierungen erstaunlicherweise eine nahezu gleiche Ausführungszeit auf und das, obwohl für die Auswertung eine unterschiedliche Anzahl Joins verwendet wurde. Diese Ergebnisse bestärkt eine Vermutung, wonach PIG bei *Joins* eine ähnliche Strategie verfolgt, wie in Kapitel 4.2 (Reduce-Side-Joins) dargelegt wurde. Weitergehende Tests waren im Rahmen dieser Arbeit leider nicht mehr durchführbar.

Anfrage 3

```
Paul_Erdoes :: r_creator(*1) > creator(*1) > r_creator(*1)
            > creator[type=equals(foaf:Person)](*1).
```

Abbildung 40: RDFPath-Anfrage 3

Bei dieser Anfrage werden zunächst alle Artikel, die „`Paul_Erdoes`“ veröffentlicht hat, betrachtet. Ausgehend von diesen Artikeln werden mit dem nächsten Lokationsschritt wieder alle Autoren bestimmt, die an den Artikeln beteiligt waren. Demzufolge hat man nun alle Co-Autoren von „`Paul_Erdoes`“ erreicht (also alle Autoren mit der *Erdös-Zahl*[37] „1“). Als nächstes erreichen die Pfade über die Rückwärtskante „`r_creator`“ wieder alle Artikel, die von den direkten Co-Autoren von „`Paul_Erdoes`“ publiziert worden sind. Mit dem letzten Lokationsschritt werden analog zu vorher wieder Autoren bestimmt. Folglich hat man eine Menge an Pfaden, die ausgehend von „`Paul_Erdoes`“ alle Autoren mit der Erdös-Zahl „`1`“ sowie alle Autoren mit der Erdös-Zahl „`2`“ erreicht. In der Ergebnismenge tauchen beide Erdös Entfernungen auf, da die Verwendung des „`*`“-Operator eine Suche nach den kürzesten Pfaden zwischen

[37] Entfernung eines Autors im Graphen mit Publikationen zum Mathematiker Paul Erdös http://www.oakland.edu/enp

„Paul_Erdoes“ und den erreichten Autoren bewirkt. Einige beispielhafte Pfade aus der Ergebnismenge sahen dabei wie folgt aus:

```
Paul_Erdoes (r_creator) Article2 (creator) Addison_Bovio

Paul_Erdoes (r_creator) Article2 (creator) Addison_Bovio
       (r_creator) Article4 (creator) Aida_Appelt
```

In Abbildung 41 sind nun die Ausführungszeiten für diese Anfrage dargestellt, wobei hier die Textvariante der Daten der enkodierten Variante gegenüber gestellt wird.

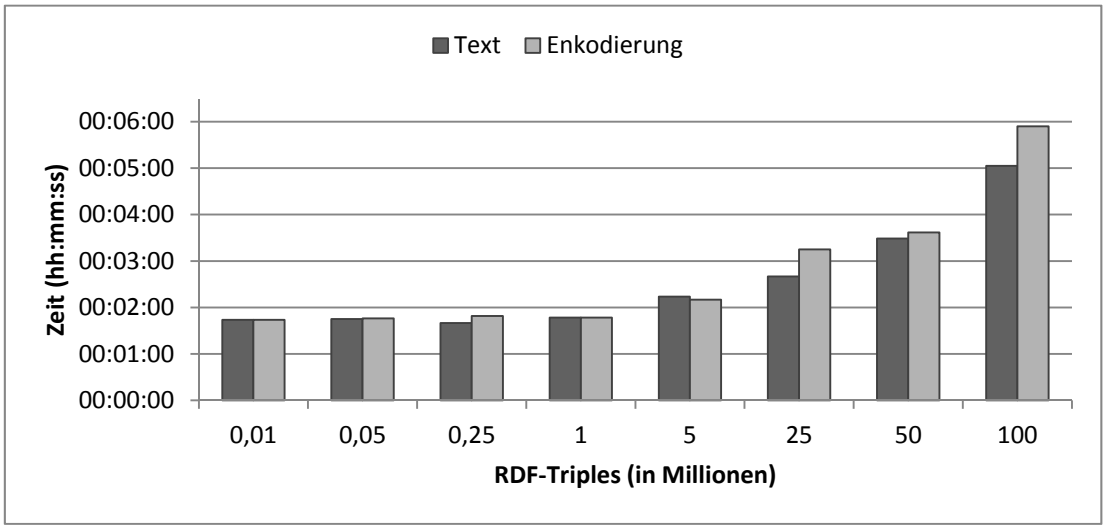


Abbildung 41: Diagramm zur Anfrage 3

In dem Diagramm ist zu sehen, dass erst ab einer Eingabegröße von fünf Millionen RDF-Triples, was ungefähr 500 Megabyte entspricht, ein Anstieg der Ausführungszeiten zu erkennen ist. Diese Beobachtung konnte auch bei mehreren weiteren Anfragen nachvollzogen werden und lässt die Schlussfolgerung zu, dass zukünftige Evaluationen, deren Anfragen eine vergleichbare Komplexität aufweisen, auf dem hier zugrunde liegenden Rechenclustern erst ab einigen Millionen RDF-Triples sinnvoll zu sein scheinen. Wichtig ist es hier abermals explizit anzumerken, dass damit keine allgemeingültige Aussage getroffen wurde, da es durchaus Anfragen geben kann, die bereits ab wenigen tausend RDF-Triples messbare Unterschiede in der Ausführungszeit aufweisen.

Weiterhin ist, wenn man die Daten in einem *Punktdiagramm* betrachtet[38], wieder ein linearer Anstieg der Ausführungszeit in der Größe der Eingabe zu beobachten. Für die Textvariante der Daten konnte er bis 1600 Millionen Triples aufgezeigt werden. Allerdings sind bereits ab fünf Millionen Triples keine weiteren Ergebnisse hinzugekommen, was bedeutet muss, dass in den größeren

[38] Im *Säulendiagramm* aus der Abbildung 41 lässt sich das nicht erkennen.

Datenbeständen keine weiteren Autoren mit einer Erdös Zahl, die kleiner gleich zwei ist, vorkommen. Folglich ist diese lineare Skalierung nicht auf die Komplexität der Anfrage übertragbar, sondern eher auf die steigende Größe der Eingabe, die bei jeder Ausführung durchlaufen werden muss, zurückzuführen. Auf den Last.fm Daten wird eine ähnliche Berechnung zu aussagekräftigeren Ergebnissen führen.

Abschließend ist in dem Diagramm außerdem noch zu erkennen, dass die enkodierten Daten wiederholt keinen Vorteil hinsichtlich der Ausführungszeit der Anfragen bewirken. Ein Blick auf die Datenmengen, die während der Shuffle-Phase durch das Netzwerk übertragen werden, zeigt allerdings, dass bei der enkodierten Variante im Durchschnitt 70% weniger Daten übertragen werden mussten. Diese Werte sind in der Abbildung 42 exemplarisch für drei unterschiedliche Eingabegrößen dargestellt.

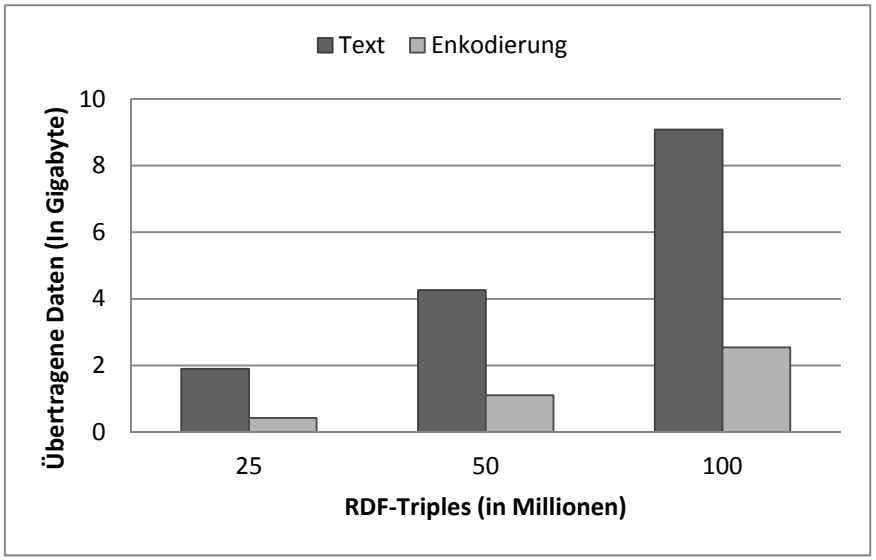


Abbildung 42: Diagramm zur Anfrage 3 (2)

Da eigentlich zu erwarten gewesen ist, dass sich eine so erhebliche Reduktion der zu übertragenden Daten positiv auf die Ausführungszeit auswirkt, mussten die Kosten, die das Dictionary Encoding während der Auswertung verursacht, höher sein als anfänglich angenommen. Unter den Kosten sind in diesem Zusammenhang Zugriffe auf das Wörterbuch zu verstehen, um einen enkodierten Knoten oder eine enkodierte Kante in ihrer ursprünglichen Repräsentationsform (Text) betrachten zu können. Dies ist beispielsweise bei der Verwendung von Filtern, bei der Erzeugung der nicht enkodierten Ergebnismengen aber auch bei der Prüfung auf Zyklen der Fall. In solchen Fällen muss für einen einzigen Pfad häufig mehrmals auf das Wörterbuch zugegriffen werden. Die Frage, die man daran anknüpfend stellen kann, ist, ob es überhaupt Anfragen gibt, in

denen sich die Verwendung des Dictionary Encodings nicht nur positiv auf den Speicherplatzbedarf auswirkt, sondern auch in einer Verbesserung der Ausführungszeit resultiert. Weiterhin war es auch interessant zu erfahren, ob sich diese Beobachtungen auch bei den realen Last.fm Daten zeigen würden oder ob es sich hier um einen Effekt handelt, der nur bei den synthetischen DBLP Daten auftritt.

5.3 Last.fm Evaluation

In diesem Abschnitt wird die Auswertung von mehreren RDFPath Anfragen, die auf den gesammelten *realen Daten des Musikdienstes Last.fm* ausgeführten werden, vorgestellt. Analog zur vorrangegangenen DBLP Evaluation werden auch hier die Anfragen auf unterschiedlichen Datengrößen ausgeführt, die bei Bedarf in Tabelle 7 nachzulesen sind. Da RDFPath insbesondere im Hinblick auf die Analyse von *sozialen Netzwerken* und *Freundschaftsbeziehungen* konzipiert wurde, war es hier deutlich einfacher geeignete Anfragen auszurücken und zu interpretieren.

Auch hier wurden die unterschiedlichen Datengrößen zunächst in ihrer ursprünglichen Textform in den RDFPath-Store eingepflegt. Sie werden im Folgenden als „**Text**“ aufgeführt. Daneben wurden sie noch mittels Dictionary Encoding in eine kompaktere Repräsentationsform gebracht und ein zweites Mal in den RDFPath-Store eingepflegt. Solche Daten werden im Folgenden als „**Enkodierung**“ bezeichnet. Die Reduktion des Speicherbedarfs lag hier, ähnlich zu den synthetischen DBLP Daten, im Durchschnitt bei ca. 65% und schwankte im Bereich zwischen 63 und 67%. Allerdings wurden hier deutlich weniger Wörterbucheinträge erzeugt, als bei den synthetischen Daten. Dies ist zum einen auf eine hohe *Konnektivität* innerhalb des RDF-Graphen zurückzuführen, der durch die Last.fm Daten repräsentiert wird und zum anderen in der relativ großen Anzahl neu erzeugter DBLP-Daten begründet, zu denen Autoren und Publikationen zählen.

Anfrage 4

```
(1)     Peter :: userKnows(*2).
(2)     Peter :: userKnows(*4).
(3)     Peter :: userKnows(*8).
(4)     Peter :: userKnows(*16).
(5)     Peter :: userKnows(*32).
```

Abbildung 43: RDFPath-Anfrage 4

Bei der Anfrage 4 (1) werden die Freundschaftsbeziehungen von „Peter“ analysiert, wobei hier nach Bekanntschaften gefragt wird, die über maximal zwei Kanten „userKnows“ erreicht werden können. Weiterhin werden durch den „*“-Operator nur die kürzesten Pfade betrachtet, d.h. wenn eine Person sowohl über 2 Kanten als auch direkt über nur eine Kante erreicht werden kann, so wird der Pfad mit nur einer Kante gewählt und der andere verworfen. Folglich setzt sich die Ergebnismenge aus Pfaden der Länge eins bis zwei zusammen, die alle von Peter ausgehen. Einige Beispiele aus der Ergebnismenge[39] sahen dabei wie folgt aus:

```
Peter (userKnows) Adam
Peter (userKnows) Adam (userKnows) Gerald
Peter (userKnows) Vicente
```

Fragestellungen dieser Art werden im Kontext von semantischen Web Technologien auch gerne als *Friend of a Friend* (FOAF) Anfragen[40] bezeichnet. Im Rahmen dieser Evaluation wurden dabei nicht nur die FOAF-Pfade mit maximal zwei Kanten betrachtet, sondern, wie in den Anfragen (2) - (5) dargestellt, auch für die maximalen Entfernungen 4, 8 , 16 und 32 ausgewertet. In Abbildung 44 werden nun die Messungen aus (1) - (5) in einem Diagramm vereint, um die Auswertungszeiten in Relation zu der maximalen FOAF-Entfernung zu setzen. Somit kann aus dieser Grafik abgelesen werden, wie sich die Auswertungszeit für eine feste Dateigröße verhält, wenn die maximale FOAF-Entfernung verändert wird. Um die Übersichtlichkeit zu steigern, wird der Verlauf für nur vier Dateigrößen mit jeweils 25, 50, 100 sowie 200 Millionen RDF-Tripels dargestellt.

[39] Die Benutzernamen wurden anonymisiert.

[40] http://www.foaf-project.org/

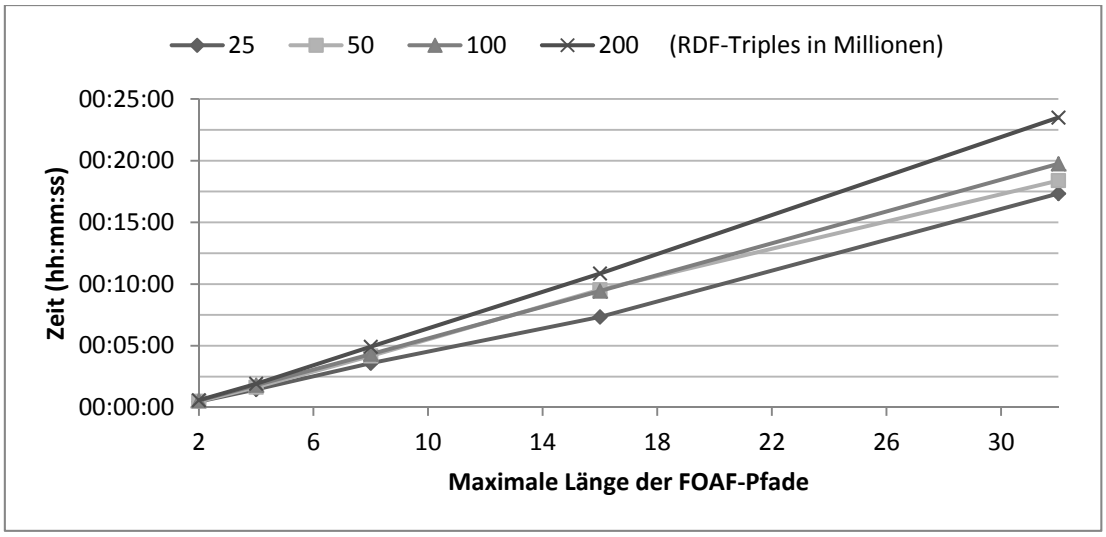


Abbildung 44: Diagramm zur Anfrage 4

In diesem Diagramm ist zu erkennen, dass die Ausführungszeiten der unterschiedlichen Dateigrößen erwartungsgemäß mit der Anzahl an RDF-Triples zunehmen, d.h. der Verlauf für die Messungen mit 200 RDF-Triples ist stets oberhalb des Verlaufs für die Messungen mit nur 100 RDF-Triples usw.

Weiterhin lässt sich aus diesem Diagramm ableiten, dass die Ausführungszeit der Anfragen linear in der Länge der betrachteten FOAF-Pfade ansteigt. Das bedeutet, dass eine hier vorgestellte FOAF-Anfrage, die doppelt so lange Pfade betrachten kann, nur linear mehr Zeit für die Ausführung benötigt. Hier wurde hingegen ein *exponentielles Wachstum* der Ausführungszeit vermutet. Eine Erklärung hierfür lieferte eine Analyse der generierten Ergebnismengen. Wenn die Anzahl von Personen, die bei den FOAF-Anfragen unterschiedlicher Länge erreicht werden, betrachtet wird, so lässt sich folgendes feststellen: Bereits mit Anfrage (3) werden im Durchschnitt rund 92% aller Personen, die in den Datenbeständen vorgekommen, mit FOAF-Pfaden erreicht. Dementsprechend enthalten auch die Ergebnismengen der Anfragen (3) - (5) eine vergleichbare Anzahl an Pfaden, was den beobachteten linearen Verlauf bei einer maximalen Pfadlänge von 8 bis 32 aufklärt. Im Vergleich dazu wurde für die Pfadlängen von 2 bis 8 ein um etwa 30% höherer linearer Anstieg der Ausführungszeit gemessen.

Darüber hinaus lässt sich hier festhalten, dass „`Peter`“ über nur 8 Kanten so gut wie jede beliebige Person kennt. Diese Zahl ist dabei nicht zufälligerweise so klein. An dieser Stelle ist das sogenannte *Kleine-Welt-Phänomen* zu beobachten. Es beschreibt eine These, wonach jeder Mensch auf der Welt mit

jedem anderen über maximal 6 FOAF-Beziehungen[41] verbunden ist [Leskovec, et al., 2008]. In Anbetracht der Tatsache, dass über den Musikdienst Last.fm sicherlich nur ein Bruchteil der realen FOAF-Beziehungen abgebildet wird, ist die Zahl Acht ein erstaunliches Resultat.

Anfrage 5

```
* :: userListenedTo > trackSimilar
     > trackTopFans [userAge=equals(26)].
```

Abbildung 45: RDFPath-Anfrage 5

Bei dieser Anfrage wurde versucht, eine möglichst große Anzahl an Zwischenergebnissen zu erzeugen. Hierzu wird ausgehend von allen Personen abgefragt, welche Musiktitel sie in letzter Zeit gehört haben. Im nächsten Schritt werden zu allen Musiktiteln mit der Kante „`trackSimilar`" die ähnlichen Werke betrachtet. Zu den somit erreichten Musiktiteln werden mit dem letzten Lokationsschritt die Lieblingsfans bestimmt. Die Menge an Pfaden wird abschließend noch dahingehend gefiltert, dass die Lieblingsfans genau 26 Jahre alt sein sollen. Ein Beispiel aus der Ergebnismenge[42] sah dabei wie folgt aus:

```
Rebekka (userListenedTo) Hans_Zimmer_-_Spider_Pig (trackSimilar)
        John_Williams_-_Main_Title (trackTopFans) Alex
```

Die Intention hinter dieser Anfrage war es, eine Hypothese in Bezug auf das Dictionary Encoding zu untersuchen. So sollte die Verwendung der implementierten Dictionary Encoding Strategie bei Anfragen mit großen Zwischenergebnissen, die sowohl im Netzwerk übertragen aber auch im HDFS abgespeichert werden mussten, Vorteile bezüglich der Ausführungszeit aufweisen. Hierfür wurde versucht mit jedem Lokationsschritt die Menge an Zwischenergebnissen möglichst schnell wachsen zu lassen und im letzten Lokationsschritt durch eine Filterung eine möglichst kompakte Ergebnismenge zu bekommen. Die kompakte Ergebnismenge sollte die Kosten für die Dekodierung der resultierenden Ergebnismenge möglichst gering halten.

In Abbildung 46 sind die Ausführungszeiten für diese Anfrage dargestellt, wobei hier die Textvariante der Daten der enkodierten Variante gegenüber gestellt wird.

[41] Auch bekannt als „Six degrees of separation"

[42] Die Benutzernamen wurden anonymisiert.

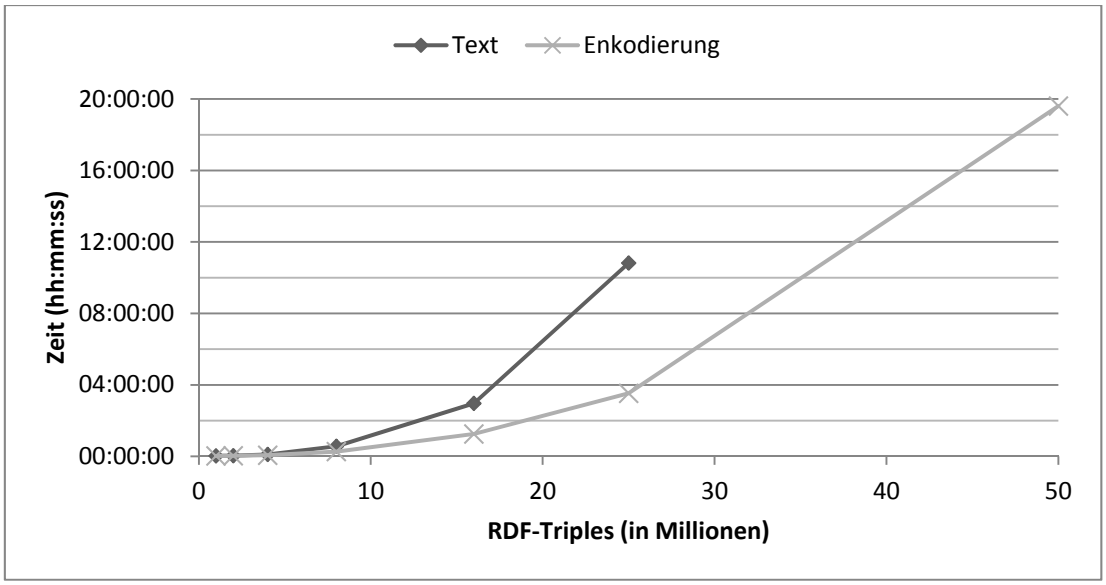


Abbildung 46: Diagramm zur Anfrage 5

Wie in Abbildung 46 zu sehen ist, hat sich die implementierte Dictionary-Encoding Strategie positiv auf die Ausführungszeiten ausgewirkt. Bei der gleichen Anzahl an RDF-Triples ist die Kurve für die enkodierte Variante stets unterhalb der Kurve für die reine Textvariante. Weiterhin ist zu sehen, dass die Auswertung auf der reinen Textvariante lediglich bis zu 25 Millionen Triples möglich gewesen ist. Bei Versuchen mit 50 Millionen Triples ist während des Auswertungsprozesses der HDFS Speicherplatz ausgegangen, was zum Abbruch der Ausführung führte. Hingegen konnte die enkodierte Variante noch auf 50 Millionen ausgewertet werden und schlug erst bei 100 Millionen RDF-Triples fehl. Daraus lässt sich schlussfolgern, dass die Verwendung der enkodierten Datenquellen die Auswertung einiger Anfragen ermöglicht, denen bei der reinen Textvariante der Speicherplatz im Rechencluster nicht ausreicht.

Als nächstes lässt sich aus dem Diagramm ableiten, dass die Ausführungszeiten der Anfrage, sowohl auf der Textvariante als auch auf den enkodierten Daten, exponentiell in Abhängigkeit der Eingabegröße (RDF-Triples) zu steigen scheinen. Dieses Wachstum lässt sich auch in Bezug auf die Speichernutzung aufzeigen. Diesbezüglich sind in der nachfolgenden Abbildung 47 vier Diagramme aufgeführt.

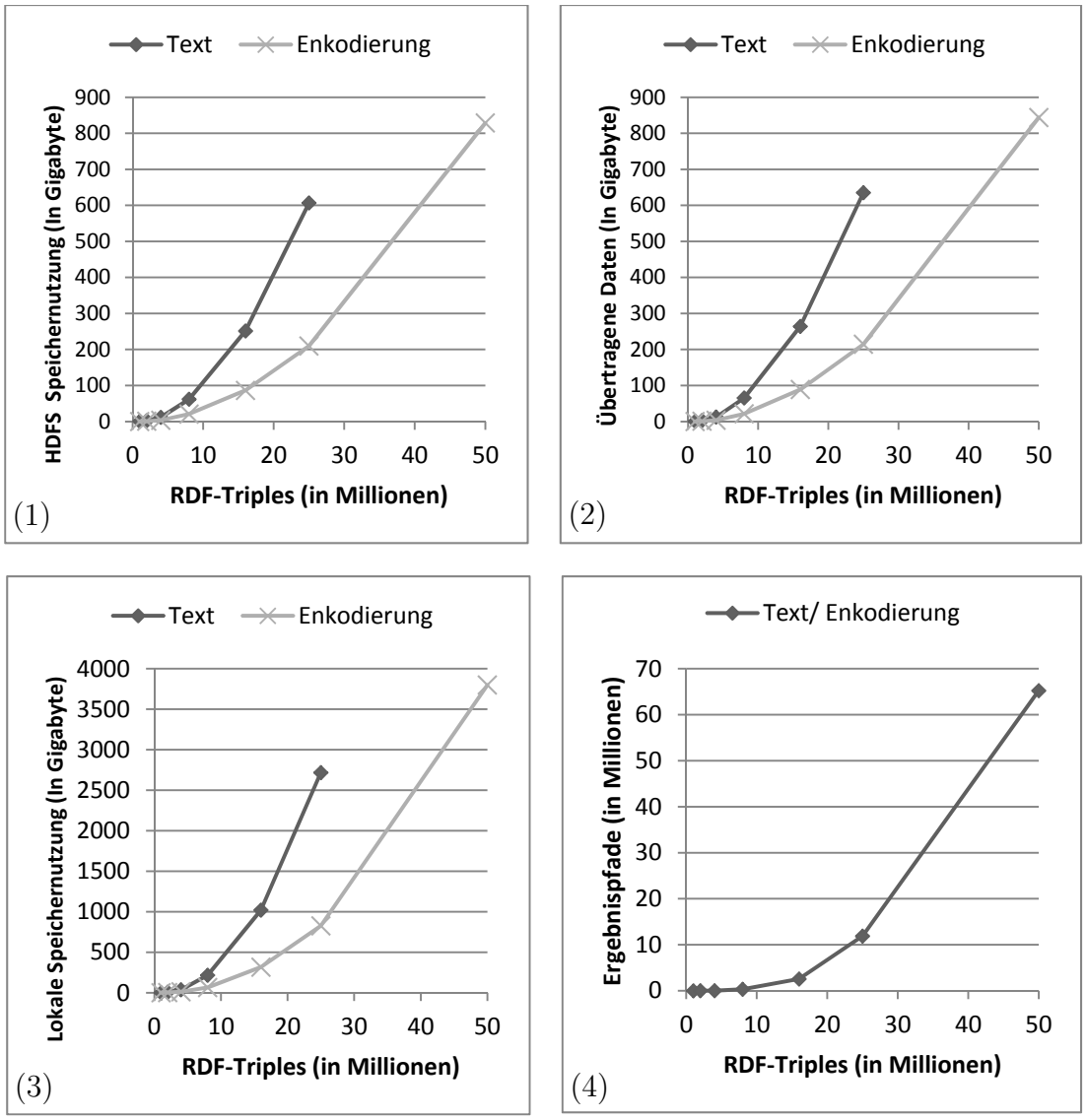


Abbildung 47: Diagramm zur Anfrage 5 (2)

Im Diagramm (1) wird der HDFS Speicherplatz, der während dem gesamten Auswertungsprozess beansprucht wurde, in Relation zu der Eingabegröße gesetzt. Die enkodierten Daten haben erwartungsgemäß weniger Speicherplatz benötigt. Weiterhin ist auch hier eine exponentielle Skalierung zu beobachten. Analog zum HDFS Speicher führt die Betrachtung des Diagrammes (2), in dem die übertragenen Daten im Netzwerk dargestellt sind, zu den gleichen Erkenntnissen. Gleiches gilt auch für das Diagramm (3). So weist auch die Nutzung des lokalen Speichers einen exponentiellen Charakter auf und die enkodierte Variante schneidet deutlich besser ab als die reine Textvariante. Ergänzend wird im vierten Diagramm die Anzahl an Pfaden, die bei der jeweiligen Eingabegröße in die Ergebnismenge aufgenommen werden, grafisch veranschaulicht.

Abschließend lässt sich hier festhalten, dass die zuvor aufgestellte Hypothese zur Dictionary Encoding Strategie nicht widerlegt werden konnte. Es konnte gezeigt werden, dass bei einer Anfrage mit großen Zwischenergebnissen, die sowohl im Netzwerk übertragen als auch im HDFS abgelegt abgespeichert werden müssen, die Verwendung der implementierten Dictionary Strategie Vorteile bezüglich der Ausführungszeit aufweist. Auch hier gilt wieder, dass die getroffenen Aussagen keinen allgemeingültigen Charakter aufweisen und vielmehr als Richtwert und weniger als eine nachgewiesene Gesetzmäßigkeit zu verstehen sind. An dieser Stelle sei noch angemerkt, dass sich ähnliche Ergebnisse auch auf den synthetischen DBLP-Daten konstruieren ließen.

An diese Erkenntnisse anknüpfend sollten mit der nächsten Anfrage unter anderem die Kosten des Dictionary Encodings genauer betrachtet werden.

Anfrage 6

```
Michael_Jackson :: artistTracks
       [trackAlbum=equals(Michael_Jackson_-_Thriller)]
       > trackSimilar [trackDuration=min(50000)]
       > trackTopFans [userCountry=equals(DE)].
```

Abbildung 48: RDFPath-Anfrage 6

Bei dieser Anfrage werden zunächst alle Musiktitel von „Michael Jackson“ bestimmt. Allerdings werden nur diejenigen weiter betrachtet, die auf dem Album „Michael Jackson - Thriller“ vorkommen. Zu den verbleibenden Musiktiteln werden mit dem nächsten Lokationsschritt alle ähnlichen Werke bestimmt, wobei diese mindestens 50 Sekunden lang sein müssen. Abschließend werden nun unter Verwendung der Kante „trackTopFans“ alle Bewunderer dieser Titel bestimmt, die ihren Wohnort in Deutschland haben. Zwei Beispiele aus der Ergebnismenge sahen dabei wie folgt aus:

```
Michael_Jackson (artistTracks) Michael_Jackson_-_Beat_It
(trackSimilar) Michael_Jackson_-_Billie_Jean (trackTopFans) Mark

Michael_Jackson (artistTracks) Michael_Jackson_-_Someone_in_the_Dark
(trackSimilar) Rihanna_-_Russian_Roulette (trackTopFans) Megan
```

Mit dieser Anfrage sollten unterschiedliche Aspekte der Implementierung untersucht werden. Zum einen sollten die Eigenschaften einer Anfrage, die auf mehrere unterschiedliche Komponenten der Implementierung zurückgreift, aufgezeigt werden. Zum anderen sollten mit dieser Anfrage, wie schon zuvor erwähnt, die Kosten des Dictionary Encodings genauer spezifiziert werden. Hierfür wurde bei der Konzeption dieser Anfrage darauf Wert gelegt, dass die zuvor

festgestellten vorteilhaften Bedingungen nur wenig Einfluss auf die Ausführungszeiten haben. Dementsprechend sollten die filternden Unteranfragen die Anzahl an Zwischenergebnisse möglichst klein halten, sodass nur wenige Daten über das Netzwerk übertragen und im HDFS abgespeichert werden müssen. Weiterhin sollten bei der Auswertung der Anfrage auch möglichst viele Zugriffe auf das Wörterbuch notwendig werden. Auch dieser Aspekt wird von den Unteranfragen abgedeckt.

In der nachfolgenden Abbildung 49 ist nun die Auswertung der gemessenen Werte dargestellt.

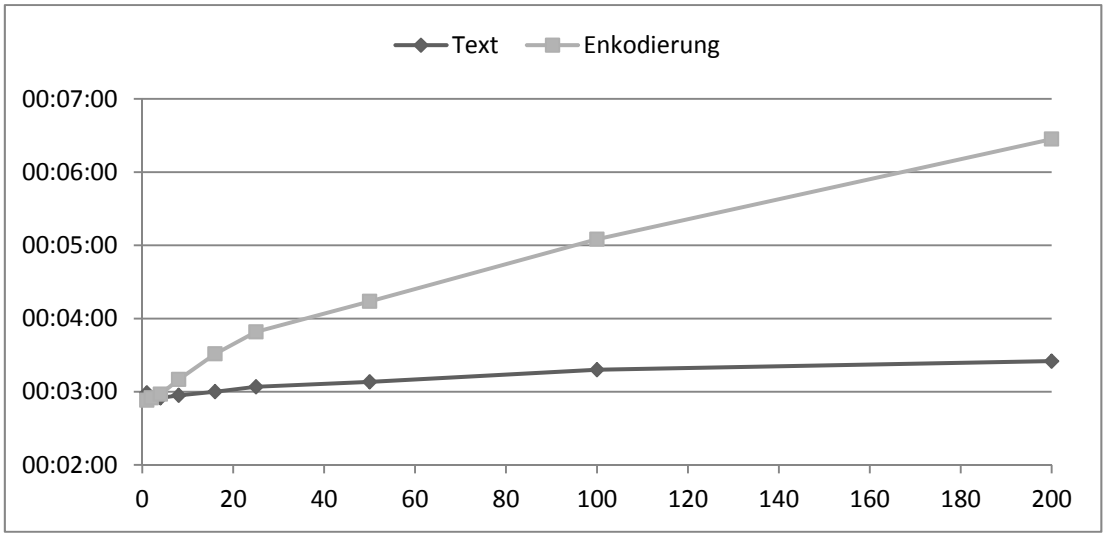


Abbildung 49: Diagramm zur Anfrage 6

Entsprechend der Erwartung hat sich gezeigt, dass die Anfrage auf den reinen Textdaten deutlich schneller ausgewertet wird als unter Verwendung des Dictionary Encodings. Die genauen Gründe hierfür lassen sich zwar nur bedingt festschreiben, die beiden zuvor erwähnten Aspekte, kleine Zwischenergebnisse und viele Wörterbuchzugriffe, beschreiben allerdings zumindest eine wesentliche Einflussgröße. Die Auswirkungen der filternden Unteranfragen sind dabei in der Abbildung 50 aufgeführt.

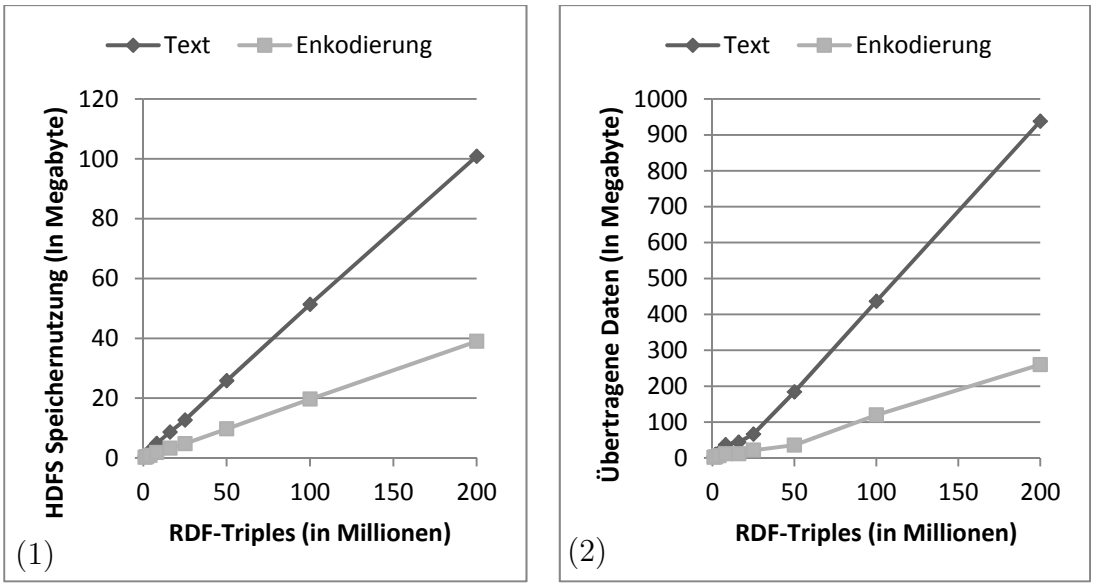


Abbildung 50: Diagramm zur Anfrage 6 (2)

Im HDFS werden lediglich 10 – 100 Megabyte (1) abgespeichert und im Netzwerk wird bei der größten Eingabe nur ein Gigabyte an Daten übertragen (2). Für einen Auswertungsprozess, der fünf Reduce-Side-Joins beinhaltet, sind das in Anbetracht des Rechencluster keine relevanten Einflussgrößen. Dementsprechend kann auch das Dictionary Encoding nicht davon profitieren, deutlich geringere Datenmengen verarbeiten zu müssen. Folglich liegt die Vermutung nahe, dass die gemessene und im Diagramm dargelegte Differenz zwischen der enkodierten und der ursprünglichen Textvariante auf die häufigen Zugriffe auf das Wörterbuch zurückzuführen ist, die wahrscheinlich einen erheblichen Einfluss auf die Ausführungszeiten haben.

Abschließend ist in Abbildung 49 zu erkennen, dass die Ausführungszeiten sowohl für die Textvariante als auch für die enkodierten Daten linear mit der Anzahl an betrachteten RDF-Triples steigen. Dabei scheint diese Anfrage, im Vergleich zu den Übrigen, besonders gut zu skalieren. Dies lässt sich zumindest teilweise auf die zahlreichen Filter zurückführen, welche die Anzahl an Zwischenergebnissen klein halten. Dies resultiert darin, dass in den Reduce-Side-Joins nur kleine Pfadmengen vereint werden müssen.

5.4 Ergebnisse

Zusammenfassend lässt sich festhalten, dass die meisten ausgewerteten Anfragen eine lineare Skalierung in Abhängigkeit der Eingabegröße (RDF-Triple) aufzuweisen scheinen. Diese lineare Abhängigkeit traf dabei in allen betrachteten Fällen auch auf die Menge an Daten zu, die zum einen im HDFS abgespeichert und zum anderen über das Netzwerk übertragen wurden. Hier muss allerdings angemerkt werden, dass die geringe Anzahl an Messpunkten keine eindeutigen Schlussfolgerungen zulässt. Bei den Anfragen, deren Ausführungszeit eher exponentiellen Charakter aufwies, zeigten die Datenmengen ebenso ein exponentielles Wachstum. Dementsprechend liegt die Vermutung nahe, dass die Auswertungszeiten maßgeblich von der Größe der zu verarbeitenden Daten dominiert werden. Zudem konnte strichprobenartig beobachtet werden, dass die Prozessorauslastung während der Auswertungsprozesse auf allen Servern relativ gering ausfiel. Folglich sind die Flaschenhälse beim Lesen, Schreiben und Übertragen der Datenmengen zu suchen. Hierbei konnte allerdings nicht zweifelsfrei geklärt werden, ob die Festplatten oder die Netzwerkverbindung den dominierenden Flaschenhalts darstellt. Es ist durchaus möglich, dass sich diese Frage nicht allgemeingültig beantwortet lässt und sich die Situation von Anfrage zu Anfrage unterscheidet. Erschwerend kam in diesem Zusammenhang hinzu, dass Hadoop nur über wenige *Monitoring-Möglichkeiten* verfügt. Standardmäßig wird lediglich ein Fortschrittsbericht in Form einer *Webschnittstelle* angeboten, der mit einem *Browser* abgerufen werden kann. Weitere Informationen werden in *Log-Dateien* abgespeichert, die auf jedem Server separat angelegt werden. Konkrete Informationen zur Auslastung der vorhandenen Kapazitäten sind, mit Ausnahme des verwendeten Speicherplatzes, nicht verfügbar.

Weiterhin hat sich gezeigt, dass eine Auswertung auf diesem Rechencluster in vielen Fällen erst ab ca. fünf Millionen RDF-Triples sinnvoll erscheint, was je nach Datenquelle ungefähr 500 Megabyte entspricht. Unterhalb dieser Werte scheint der Overhead für die Koordination und Ausführung von MapReduce Aufträgen höher als der Nutzen zu sein, der aus der verteilten Analyse der RDF-Daten resultiert. Betrachtet man hingegen die obere Grenze, so stellt man fest, dass für viele Anfragen sogar 1600 Millionen RDF-Triples kein Problem dargestellt haben. Hier muss allerdings berücksichtigt werden, dass durch die vertikale Partitionierung nur ein Bruchteil der Daten tatsächlich gelesen und verarbeitet werden musste. Nichtsdestotrotz lässt sich festhalten, dass Analysen von 1600 Millionen Triples in den meisten Fällen problemlos möglich waren. Eine Ausnahme hiervon bildeten die Anfragen mit exponentiellem Wachstum der Daten, für die nicht genügend Speicherplatz verfügbar war.

Dennoch darf angenommen werden, dass auch eine größere Anzahl an RDF-Triples mit dem hier zugrunde liegenden Rechencluster verarbeitet werden könnte. Zugleich sollten sich größere Rechencluster, gemäß der aufgezeigten linearen Skalierung vieler Anfragen, positiv auf die Ausführungszeiten und auf die maximale Größe der zu verarbeitenden Datenmengen auswirken.

Zudem wurde aufgezeigt, dass die enkodierte Repräsentationsform der Datenbestände erwartungsgemäß deutlich weniger Speicherplatz benötigt, als die ursprüngliche Textform. Durch das Dictionary Encoding wird der Speicherplatzbedarf im Durchschnitt um ca. 65% gesenkt. Bei den Zwischenergebnissen und den im Netzwerk übertragenen Daten lag die Reduktion des Datenaufkommens in manchen Fällen sogar bei 70 - 80%. Diese vielversprechenden Zahlen wirkten sich allerdings nur eingeschränkt positiv auf die Ausführungszeiten aus. So konnten zwar Anfragen, bei denen die Zwischenergebnisse exponentiell angewachsen waren, sehr deutlich vom Dictionary Encoding profitieren. Bei den meisten anderen Anfragen hingegen, deren Datenaufkommen nur linear zunahm, waren durch die Verwendung der enkodierten Daten keine Vorteile bezüglich der Ausführungszeit zu beobachten. Für mehrere weitere Anfragen, die sich durch eine geringe Größe der Zwischenergebnisse und durch häufige Wörterbuchzugriffe charakterisieren ließen, lag die Ausführungszeit sogar deutlich unter der gemessenen Zeit für die Textvariante. Die Gründe hierfür wurden primär häufigen Wörterbuchzugriffen zugeschrieben. So muss beispielweise jeder Pfad aus der erzeugten Ergebnismenge im letzten Schritt des Auswertungsprozesses in seine ursprüngliche, dekodierte Repräsentationsform gebracht werden, was mitunter sehr viele Wörterbuchzugriffe zur Folge haben kann.

Weiterhin muss an dieser Stelle auch ein kritischer Blick auf die verwendete Datenstruktur, in diesem Fall also die Berkeley Database, geworfen werden. Es kann angenommen werden, dass hier weitere Optimierungsschritte oder gänzlich andere Ansätze zu einer Verbesserung der Ausführungszeit führen würden. Ebenso wenig kann ausgeschlossen werden, dass hier insbesondere die Festplatte den Flaschenhals darstellte. Die vielen zufälligen Festplattenzugriffe des Dictionary Encodings, die parallel zum Auswertungsprozess stattfanden, haben die Lese-/Schreibgeschwindigkeit der Reducer und Mapper sicherlich negativ beeinflusst. Der Fokus dieser Arbeit lag allerdings auf der Konzeption und der Implementierung von verteilten Lösungsansätzen für die Analyse von RDF-Graphen, sodass nur vergleichsweise wenig Zeit in die Betrachtung unterschiedlicher Dictionary Encoding Strategien investiert werden konnte.

Weiterhin konnte gezeigt werden, dass die Pfadanfragesprache RDFPath ein nützliches Werkzeug für die Analyse von RDF-Graphen darstellt. So konnten

im Rahmen der Evaluation viele interessante Fragestellungen aus der Graphen Theorie aufgriffen und durch entsprechende RDFPath Anfragen beantwortet werden. Hierbei hat sich beispielsweise gezeigt, dass in den realen Musikdaten von Last.fm das kleine Welt Phänomen zu beobachten ist. Analog dazu konnten auf den synthetischen DBLP Daten Aussagen über die Erdös-Zahl getroffen werden. Wobei hier natürlich nicht der echte Mathematiker Paul Erdös sondern ein synthetisch generierter Autor, der nur zufälligerweise denselben Namen aufweist, betrachtet wurde. Auch wenn die wesentlichen Komponenten, wie die Suche nach den kürzesten Pfaden, bereits ausgiebig getestet und dargelegt wurden, so konnten viele Ausdrucksmöglichkeiten, wie beispielsweise die unterschiedlichen Ergebnisformate, im Rahmen dieser Arbeit nur bruchstückhaft ausprobiert werden.

Abschließend sollten noch einige Worte zu den zwei betrachteten Datenquellen angemerkt werden. So konnten alle Erkenntnisse, die auf die implementierten Strategien und Konzepte von RDFPath zurückzuführen sind, auf beiden Datenbeständen nachvollzogen werden. Nichtsdestotrotz lässt sich an dieser Stelle festhalten, dass die Verwendung der realen Daten von Last.fm gegenüber den synthetischen klar vorzuziehen ist. Auch wenn die Beschaffung der Daten mit einem zusätzlichen Aufwand verbunden ist, so rechtfertigen die frei zusammenstellbaren Untergraphen und realen Charakteristika diesen mehr als deutlich. Hier zeigt sich, dass RDFPath vor allem im Hinblick auf die Analyse von sozialen Netzwerken konzipiert wurde und dementsprechend derartige RDF-Graphen einfacher zu analysieren vermag.

6 Verwandte Arbeiten

Im Folgenden werden die Gemeinsamkeiten und Unterschiede zu einigen angrenzenden Arbeiten und Projekten vorgestellt, bei denen ähnliche Ziele angestrebt oder vergleichbare Strategien gewählt wurden.

RDF Triple Stores

Im Allgemeinen umfasst ein *RDF Triple Store* die Verwaltung der Datenbestände sowie die Möglichkeit mit einer Sprache Anfragen auszudrücken. In diesem Sinne kann auch die hier zugrunde liegende Implementierung als ein Triple Store verstanden werden, allerdings lag hier der Schwerpunkt nicht auf allgemeinen Anfragen, wie sie beispielsweise mit SPARQL ausgedrückt werden können, sondern auf Pfadanfragen. Verwandte Projekte, die ebenso einen RDF Triple Store auf Grundlage von Hadoop entwickeln sind das 2008 angekündigte *Heart-Projekt* [Yoon, et al., 2009] und das im Jahre 2010 vorgestellte *SHARD* [Raytheon BBN Technologies, 2010]. Im Gegensatz zur RDFPath Implementierung, soll das *Heart-Projekt* (Highly Extensible & Accumulative RDF Table) nicht direkt auf das HDFS aufsetzen, sondern HBase verwenden. Die Entwicklung scheint allerdings aufgrund von Komplikationen in Verbindung mit HBase zurzeit nicht fortgeführt zu werden. Bei *SHARD* (Scalable, High-Performance, Robust and Distributed) handelt es sich um einen vielversprechenden Ansatz, der eigenen Aussagen zufolge bereits erfolgreich getestet wurde. Allerdings wurde bisher nur ein einziger Blogeintrag veröffentlicht, der lediglich die Grundidee konzeptuell vorstellt. Ebenso sind auch bei einigen bereits etablierten RDF Tripe Stores wie beispielsweise *Bigdata*, die ihre Wurzeln in Einzelrechnerlösungen haben, erfolgreiche Hadoop Ansätze zu beobachten [SYSTAP LLC, 2009].

Anfragesprachen für RDF

Für eine Abgrenzung zu *XPath*, *SPARQL, nSPARQL* sowie *pathRDF* sei auf den Abschnitt 3.2 verwiesen. Bei der Einführung von RDFPath wurden die Gemeinsamkeiten und Unterschiede zu vielen verwandten Sprachen ausführlich besprochen. Dementsprechend wird an dieser Stelle nicht wiederholt darauf eingegangen. Ergänzend sei hier auf eine kurze Auflistung von Pfadsprachen aus dem Jahre 2007 verwiesen, die unter [W3C community wiki, 2007] verfügbar ist.

RDF Processing mit Hadoop

Unter dem Begriff *RDF Processing* lassen sich Techniken zusammenfassen, die sich im weitesten Sinne mit der Verarbeitung von RDF-Daten auseinandersetzen. So wurde parallel zu dieser Arbeit von Herrn Alexander Schätzle eine Übersetzung von SPARQL nach Pig Latin entwickelt (*PigSPARQL*), die es ermöglicht, SPARQL Anfragen auf Hadoop auszuwerten [Schätzle, 2010]. Wie schon in Kapitel 3.2 dargelegt, verfügt SPARQL jedoch nur über eingeschränkte Navigationsmöglichkeiten [Pérez, et al., 2008]. Dementsprechend kann RDFPath auch als eine Ergänzung zu SPARQL, die zusätzliche *Navigationsmöglichkeiten* bereitstellt, vorgeschlagen werden. Eine zukünftige Zusammenführung beider Arbeiten wird daher nicht ausgeschlossen.

Weiterhin werden auch mit *PIG* Möglichkeiten bereitgestellt, beliebige Daten, zu denen auch RDF-Dokumente zählen, zu verarbeiten. Im Gegensatz zu RDFPath enthält PIG allerdings keinen eigenen Datenspeicher wie den RDF-Path-Store, der darauf ausgelegt ist Pfadanfragen, möglichst effizient auszuwerten [Olston, et al., 2008]. Mit *RDFgrid* [Bendiken, 2010] und *WebPIE* [Urbani, et al., 2010] sind noch zwei weitere angrenzende Arbeiten zu erwähnen, die ebenfalls auf Hadoop beruhen. Bei RDFgrid handelt es sich allerdings nur um ein einfaches Framework zur Verarbeitung von RDF-Daten im N-Triples Format. Eine Anfragesprache wird nicht angeboten. WebPIE (Web-scale Parallel Inference Engine) hingegen verfolgt das Ziel, *RDFS* und *OWL Reasoning* auf Hadoop zu parallelisieren.

Distributed Computing

Über die hier vorgestellte MapReduce Implementierung von Hadoop hinaus, gibt es noch einige weitere interessante Ansätze, die ebenso das Ziel verfolgen, Berechnungen automatisch zu parallelisieren und Daten zu verteilen. An erster Stelle sei natürlich die ursprüngliche MapReduce Implementierung von Google erwähnt, die allerdings nur intern bei Google verfügbar ist [Dean, et al., 2004]. Darüber hinaus gibt es noch einige weitere MapReduce Implementierungen, wobei Hadoop hier die am weitesten entwickelte Lösung anbietet. Weiterhin wird mit *Dryad* auch bei Microsoft ein Framework entwickelt, das genauso wie MapReduce, das Programmieren verteilter Anwendungen erleichtern soll [Isard, et al., 2007].

7 Zusammenfassung & Ausblick

Gegenstand dieser Arbeit war die Frage, ob eine Analyse von großen RDF-Graphen unter Verwendung von Hadoop's MapReduce Implementierung effizient durchgeführt werden kann. Um das zu beantworten, wurde ein System entworfen und implementiert, welches Pfadanfragen im MapReduce Framework verteilt auswertet. Einen zentralen Bestandteil dieser Implementierung stellt die Pfadanfragesprache RDFPath dar, die es ermöglicht, unterschiedliche Fragestellungen auf den Daten zu beantworten. Weiterhin wurde mit dem RDFPath-Store ein Datenkonzept vorgeschlagen, welches auf der vertikalen Partitionierung von RDF-Dokumenten beruht. Die partitionierten Daten werden dabei in einem eigens konzipierten Format im HDFS abgelegt. Ferner wurde mit dem Reduce-Side-Join die zentrale Komponente des Auswertungsprozesses besprochen und aufgezeigt, wie Anfragen aus der Pfadanfragesprache RDFPath in eine Sequenz von MapReduce Aufträgen abgebildet werden können. Ebenso wurde auch das Einpflegen neuer RDF-Graphen in den RDFPath-Store besprochen und die implementierte Dictionary Encoding Strategie vorgestellt. Abschließend wurde eine Evaluation auf zwei unterschiedlichen Datenbeständen durchgeführt. Hierfür wurden zum einen die synthetischen Daten, die unter Verwendung des SP^{2}Bench Generators entstanden sind, betrachtet. Zum anderen wurden reale Daten des Musikdienstes Last.fm, die mit Hilfe eines Webcrawlers gesammelt wurden, verwendet.

Zusammenfassend lässt sich hier festhalten, dass eine verteilte Analyse von RDF-Graphen unter Verwendung von RDFPath als Analysewerkzeug sinnvoll durchgeführt werden kann. Es hat sich gezeigt, dass die meisten Anfragearten – innerhalb der betrachteten Datenmengen – linear in Abhängig der Eingabegröße skalieren. Viele interessante Fragestellungen konnten mit RDFPath ausgedrückt und verteilt auf dem gesamten Rechencluster ausgewertet werden. Die hierfür verwendete MapReduce Implementierung von Hadoop hat sich dabei als eine solide Grundbasis erwiesen. Gleiches gilt für Hadoop's HDFS Implementierung und das Zusammenspiel mit dem MapReduce-Framework. Die zur Verfügung gestellte Abstraktionsschicht, also die Mapper und Reducer, waren zwar gewöhnungsbedürftig, jedoch sinnvoll einsetzbar. Allerdings hat sich die Verwendung der neuen MapReduce API, die mit Hadoop 0.20.0 eingepflegt wurde, als problematisch erwiesen. Mehrere wichtige Bibliotheken waren zum Zeitpunkt der Programmierung noch nicht vollständig portiert. Als letzte

Schlussbemerkung sei noch angeführt, dass für zukünftige Entwicklungen zusätzliche *Monitoring* und *Debugging* Möglichkeiten in Hadoop eine große Hilfe darstellen würden.

Über die hier vorgestellte Implementierung von RDFPath hinaus sind noch zahlreiche Erweiterungs- sowie Optimierungsansätze vorstellbar. Nachfolgend wird ein kurzer Ausblick, der in vier Bereiche gegliedert wurde, konzeptuell vorgestellt.

(1) **RDFPath-Store**: Da sich in diesem Zusammenhang die vertikale Partitionierung als sehr nützlich erwiesen hat, wäre es durchaus vorstellbar, dass noch granularer strukturierte RDF-Dokumente die zu verarbeitende Menge an Daten weiter senken könnten. Ebenso stellen *Indexe*, die in anderen Triple-Stores entscheidend zur Leistungsfähigkeit des Auswertungsprozesses beitragen, sicherlich auch in einem verteilt arbeitenden Framework einen interessanten Ansatz dar. Eine einfache Möglichkeit, um das Datenaufkommen noch weiter zu reduzieren, könnte auch die *Kompression* von Zwischenergebnissen darstellen. Entsprechende Mechanismen sind bereits in Hadoop implementiert und könnten auch in Verbindung mit RDFPath genutzt werden. Weiterhin seien mit der *Prematerialisierung* von Pfaden oder dem *Caching* von Zwischenergebnissen zwei Techniken erwähnt, die durch zusätzlichen Speicherplatzbedarf einen positiven Einfluss auf die Ausführungszeiten haben könnten. Darüber hinaus könnte auch die Verwendung von HBase oder *Cassandra*, zwei Speichermöglichkeiten für das MapReduce-Framework, die zufällige Lesezugriffe unterstützen, in Betracht gezogen werden.

(2) **Joins**: Über die hier vorgestellten Reduce-Side-Joins hinaus gibt es für MapReduce weitere Ansätze, die es ermöglichen Datenmengen zu vereinen. Eine Möglichkeit besteht in der Verwendung der bereits kurz angesprochenen *Map-Side-Joins*. Da hierbei deutlich weniger Daten über das Netzwerk verschickt werden müssen, weisen diese Joins im Regelfall deutlich bessere Laufzeiten auf. Nachteilig ist jedoch, dass die Verwendung von Map-Side-Joins an weitreichende Bedingungen geknüpft ist, die sich in vielen Fällen nicht erfüllen lassen. Allerdings ist ein eingeschränkter Einsatz für einige spezielle Situationen sehr wohl zu realisieren. Ebenso könnten auch die von PIG eingesetzten *Multi-Joins* betrachtet werden, die unter gewissen Voraussetzungen mehr als nur zwei Datenmengen vereinen.

(3) **Dictionary Encoding**: Die hier vorgestellte Strategie zum Dictionary Encoding weist durchaus viel Spielraum für Optimierungen auf. So könnte der Einsatz von mehreren Festplatten aber auch die Wahl effizienterer bzw. mehr der Problemstellung entsprechender Datenstrukturen den Nutzen des Dictionary Encodings weiter erhöhen. Weiterhin wäre es auch interessant, *hybride* Ansätze zu verfolgen, die beispielsweise nur besonders häufig vorkommende Bezeichner enkodieren, um die Zugriffe auf das Wörterbuch möglichst gering zu halten.

(4) **RDFPath**: Über die hier vorgestellten Werkzeuge zur Analyse von RDF-Graphen hinaus wäre es möglich, weitere Fragestellungen aus der *Graphen Theorie* zu betrachten und in die Pfadanfragesprache einfließen zu lassen. Auch ist eine Verknüpfung mit SPARQL nicht abwegig, wie beispielsweise mit den Ansätzen in [Pérez, et al., 2008] demonstriert wurde.

Literaturverzeichnis

Abadi, Daniel J., et al. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. *Proceedings of the 33rd international conference on Very large data bases, September 23-27, 2007, Vienna, Austria .* 2007.

Alkhateeb, Faisal, Baget, Jean-François und Euzenat, Jérôme. 2009. Extending SPARQL with Regular Expression Patterns (for Querying RDF). *Journal of Web Semantics, Volume 7, Number 2, April 2009.* 2009.

Azza Abouzeid, Kamil BajdaPawlikowski, Daniel Abadi, Avi Silberschatz, Alexander Rasin. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment, Volume 2, Number 1, August 2009, 922-933.* 2009.

Bendiken, Arto. 2010. RDFgrid: Map/Reduce-based Linked Data Processing with Hadoop. [Online] März 2010. [Zitat vom: 17. Juni 2010.] http://rdfgrid.rubyforge.org/.

Champin, Pierre-Antoine. 2001. *RDF Tutorial.* 2001.

Cloudera. 2010. Hadoop Training. [Online] Januar 2010. [Zitat vom: 11. März 2010.] http://www.cloudera.com/resources/?type=Training.

Dean, Jeffrey und Ghemawat, Sanjay. 2004. MapReduce: simplified data processing on large clusters. *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, p.10-10, Dezember 06-08, 2004, San Francisco, CA .* 2004.

Eickler, André, Gerlhof, Csrsten A. und Kossmann, Donald. 1995. A Performance Evaluation of OID Mapping Techniques. *Proceedings of the 21th International Conference on Very Large Data Bases, p.18-29, September 11-15, 1995 .* 1995.

Ghemawat, Sanjay, Gobioff, Howard und Leung, Shun-Tak. 2003. The Google File System. *Proceedings of the nineteenth ACM symposium on Operating systems principles, October 19-22, 2003, Bolton Landing, NY, USA.* 2003.

Isard, Michael, et al. 2007. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *Proceedings of the 2nd ACM*

SIGOPS/EuroSys European Conference on Computer Systems 2007, March 21-23, 2007, Lisbon, Portugal. 2007.

Knuth, Donald. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Reading, Massachusetts : Addison-Wesley, 1998. ISBN 0-201-89685-0.

Lausen, Georg. 2005. *Datenbanken: Grundlagen und XML-Technologien.* Freiburg : Elsevier, 2005. ISBN 3-8274-1488-1.

Leskovec, Jure und Horvitz, Eric. 2008. Planetary-scale views on a large instant-messaging network. *Proceeding of the 17th international conference on World Wide Web, April 21-25, 2008, Beijing, China .* 2008.

Manola, Frank, Miller, Eric und McBride, Brian. 2004. RDF Primer. *World Wide Web Consortium, W3C Recommendation 10 February 2004.* [Online] Februar 2004. [Zitat vom: 27. März 2010.] http://www.w3.org/TR/rdf-primer/.

Neumann, Thomas und Weikum, Gerhard. 2008. RDF3X: a RISCstyle Engine for RDF. *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), Auckland, New Zealand.* 2008.

Olston, Christopher, et al. 2008. PigLatin: A Not-So-Foreign Language for Data Processing. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada.* Juni 2008.

Oracle. 2010. Oracle Berkeley DB - Programmer's Reference Guide. [Online] 2010. [Zitat vom: 17. April 2010.] http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/BDB_Prog_Reference.pdf.

Pérez, Jorge, Arenas, Marcelo und Gutierrez, Claudio. 2008. nSPARQL: A Navigational Language for RDF. *Proceedings of the 7th International Conference on The Semantic Web, October 26-30, 2008, Karlsruhe, Germany.* 2008.

Powers, Shelley. 2003. *Practical RDF.* Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2003. ISBN: 0-596-00263-7.

Radicati Group, Inc. 2009. *Email Statistics Report, 2009-2013.* Kanada : http://www.radicati.com/?p=3229, 2009.

Raytheon BBN Technologies. 2010. Cloudera Blog. [Online] März 2010. [Zitat vom: 23. März 2010.] http://www.cloudera.com/blog/2010/03/how-raytheon-researchers-are-using-hadoop-to-build-a-scalable-distributed-triple-store/.

Schätzle, Alexander. 2010. PigSPARQL: Eine Übersetzung von SPARQL nach Pig Latin. *Masterarbeit, Albert-Ludwigs-Universität Freiburg, Technische Fakultät, Institut für Informatik, Lehrstuhl für Datenbanken und Informationssysteme.* 2010.

Schmidt, Eric, et al. 2010. Popular Techonomy: Can the world be turned into a technomic direction? *Techonomy Conference 2010, Lake Tahoe, California, USA.* [Online] August 2010. [Zitat vom: 10. August 2010.] http://link.brightcove.com/services/player/bcpid589138719001?bclid=8767563 9001&bctid=585116891001. http://techonomy.com/videos/.

Schmidt, Michael, et al. 2009. SP2Bench: A SPARQL Performance Benchmark. *Proceedings of the 2009 IEEE International Conference on Data Engineering, p.222-233, March 29-April 02.* 2009.

SYSTAP LLC. 2009. Bigdata: Scale-out Architecture. *Whitepaper, Draft.* 2009.

Urbani, Jacopo, et al. 2010. WebPIE: a Web-scale Parallel Inference Engine. *SCALE 2010 Submission.* 2010.

W3C community wiki. 2007. RDF Path Languages And Templating. [Online] Oktober 2007. [Zitat vom: 25. Juli 2010.] http://esw.w3.org/RdfPath.

Wadler, Philip. 2000. A formal semantics of patterns in XSLT. *Markup Languages, v.2 n.2, p.183-202, Spring 2000.* 2000.

—. **1999.** Two Semantics for XPath. *Draft paper.* 1999.

White, Tom. 2009. *Hadoop: The Definitive Guide.* s.l. : O'Reilly Media, Inc., 2009. ISBN: 978-0-596-52197-4.

Yoon, Edward J. und Choi, Hyunsik. 2009. Heart (Highly Extensible & Accumulative RDF Table). [Online] 2009. [Zitat vom: 23. April 2010.] http://wiki.apache.org/incubator/HeartProposal.

Ziegler, Cai-Nicolas. 2010. Social Networks and Small-Worlds: Characteristics and distinctive features. *Habilitationsvortrag, Albert-Ludwigs-Universität Freiburg.* 2010.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift