REGELBASIERTES REASONING AUF MASSIV PARALLELER HARDWARE

MARTIN PETERS

Dissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

(Dr. rer. nat.)

am Fachbereich Elektrotechnik/Informatik

der Universität Kassel

Juli 2015

Beurteilung:	Prof. Dr. Albert Zündorf Universität Kassel
	Prof. Dr. Sabine Sachweh Fachhochschule Dortmund
Tag der Disputation:	15.09.2015

Martin Peters: *Regelbasiertes Reasoning auf massiv paralleler Hardware,* © Juli 2015

ZUSAMMENFASSUNG

Eine wesentliche Funktionalität bei der Verwendung semantischer Technologien besteht in dem als Reasoning bezeichneten Prozess des Ableitens von impliziten Fakten aus einer explizit gegebenen Wissensbasis. Der Vorgang des Reasonings stellt vor dem Hintergrund der stetig wachsenden Menge an (semantischen) Informationen zunehmend eine Herausforderung in Bezug auf die notwendigen Ressourcen sowie der Ausführungsgeschwindigkeit dar. Um diesen Herausforderungen zu begegnen, adressiert die vorliegende Arbeit das Reasoning durch eine massive Parallelisierung der zugrunde liegenden Algorithmen und der Einführung von Konzepten für eine ressourceneffiziente Ausführung. Diese Ziele werden unter Berücksichtigung der Verwendung eines regelbasierten Systems verfolgt, dass im Gegensatz zur Implementierung einer festen Semantik die Definition der anzuwendenden Ableitungsregeln während der Laufzeit erlaubt und so eine größere Flexibilität bei der Nutzung des Systems bietet.

Ausgehend von einer Betrachtung der Grundlagen des Reasonings und den verwandten Arbeiten aus den Bereichen des parallelen sowie des regelbasierten Reasonings werden zunächst die Funktionsweise von Production Systems sowie die dazu bereits existierenden Ansätze für die Optimierung und im Speziellen der Parallelisierung betrachtet. Production Systems beschreiben die grundlegende Funktionalität der regelbasierten Verarbeitung und sind somit auch die Ausgangsbasis für den RETE-Algorithmus, der zur Erreichung der Zielsetzung der vorliegenden Arbeit parallelisiert und für die Ausführung auf Grafikprozessoren (GPUs) vorbereitet wird. Im Gegensatz zu bestehenden Ansätzen unterscheidet sich die Parallelisierung insbesondere durch die gewählte Granularität, die nicht durch die anzuwendenden Regeln, sondern von den Eingabedaten bestimmt wird und sich damit an der Zielarchitektur orientiert.

Aufbauend auf dem Konzept der parallelen Ausführung des RETE-Algorithmus werden Methoden der Partitionierung und Verteilung der Arbeitslast eingeführt, die zusammen mit Konzepten der Datenkomprimierung sowie der Verteilung von Daten zwischen Haupt- und Festplattenspeicher ein Reasoning über Datensätze mit mehreren Milliarden Fakten auf einzelnen Rechnern erlauben. Eine Evaluation der eingeführten Konzepte durch eine prototypische Implementierung zeigt für die adressierten leichtgewichtigen Ontologiesprachen einerseits die Möglichkeit des Reasonings über eine Milliarde Fakten auf einem Laptop, was durch die Reduzierung des Speicherbedarfs um rund 90% ermöglicht wird. Andererseits kann der dabei erzielte Durchsatz mit aktuellen State of the Art Reasonern verglichen werden, die eine Vielzahl an Rechnern in einem Cluster verwenden.

ABSTRACT

One key feature when using semantic technologies is the use of reasoners to infer new knowledge by materializing facts that are implicitly given by a knowledge base. Considering the growing amount of available semantic data, the reasoning process becomes challenging with respect to the resource efficiency as well as to the speed of execution. To address these issues, this thesis focuses on gaining performance improvements by a massive parallelization of a rule-based reasoning process and resource optimizations by data compression and distribution between main memory and the hard disk. The use of a rule-based approach allows the definition of inference rules during runtime instead of implementing static inference rules, which gives a greater flexibility for the use of the system. Thus, the objective of this work is the conception and implementation of a rule-based reasoner that uses massively parallel hardware like Graphics Processor Units (GPUs) and is able to reason on datasets with several billions of facts on a single computer.

Based on an analyses of the related work with a particular focus on rule-based and parallel reasoning, fundamentals of production systems as well as existing concepts for a parallel implementation are presented. Production systems describe the essential functionality of a rule-based execution and thus are also the basis for the RETE algorithm, which is used in this thesis for a parallelization and preparation for an execution on GPUs to achieve the defined objectives. In contrast to existing approaches of parallelizing the RETE algorithm, the introduced concepts in this thesis especially differ in the chosen granularity of parallelization, which is influenced by the input data instead of the given rules. This leads to a much more fine-grained parallelization that can efficiently be executed on the target architecture.

The introduction of a parallel RETE execution is followed by the proposing of new concepts for partitioning and distributing the workload of the reasoning process. In conjunction with methods of data compression and data distribution between the main memory and the hard disk, these concepts allow the reasoning on several billion triples on a single machine. An evaluation of the introduced concepts using a prototypical Java implementation shows, that the main memory consumption for the addressed lightweight rule-based reasoning can be reduced by about 90%. This not only makes it possible to apply reasoning on about one billion statements using only a single laptop, but also to achieve a throughput that is comparable or even higher than state of the art reasoners using multiple computing nodes in a cluster.

INHALTSVERZEICHNIS

1 EINLEITUNG 1
- 1.1 Problembeschreibung 2
- 1.2 Zielsetzung und Forschungsfragen 4
- 1.3 Forschungsgegenstand 5
- 1.4 Aufbau der Arbeit 6

I GRUNDLAGEN UND VERWANDTE ARBEITEN 8

2 SEMANTISCHES WEB 9
- 2.1 Einführung in das semantische Web 9
- 2.2 Resource Description Framework (RDF) 12
- 2.3 RDF-Schema 14
- 2.4 Web-Ontologiesprachen 16
- 2.5 Linked Data 17

3 WISSENSGENERIERUNG DURCH REASONING 20
- 3.1 Reasoning Grundlagen 20
  - 3.1.1 Komplexität 22
  - 3.1.2 Beschreibungslogik 22
- 3.2 Reasoning-Mechanismen 24
  - 3.2.1 Tableau Reasoning 24
  - 3.2.2 Regelbasiertes Reasoning 25
- 3.3 Verwandte Arbeiten 27
  - 3.3.1 Regelbasiertes Reasoning 27
  - 3.3.2 Paralleles Reasoning 29
  - 3.3.3 Limitierung bestehender Reasoner-Ansätze 34
  - 3.3.4 Übertragung der Erkenntnisse auf die Forschungsfragen 36

4 PRODUCTION SYSTEMS ALGORITHMEN 37
- 4.1 Regeln und Regel-Syntax 38
- 4.2 RETE-Algorithmus 39
  - 4.2.1 RETE-Netz 40
  - 4.2.2 Pattern-Matching 41
  - 4.2.3 Regelausführung 43
- 4.3 Optimierungen und Erweiterungen des RETE-Algorithmus 44
  - 4.3.1 Optimierungen basierend auf der Netzwerkgestaltung 44
  - 4.3.2 Optimierungen basierend auf einer erweiterten Behandlung von RETE-Knoten 48
  - 4.3.3 Zusätzliche Optimierungen und Erweiterungen 50

5 PARALLELISIERUNG VON PRODUCTION SYSTEMS 51
5.1 Regel-Partitionierung 51
5.2 Daten-Partitionierung 53
5.3 Weitere Ansätze 54

II REASONING AUF MASSIV PARALLELER HARDWARE 56

6 PROGRAMMIERUNG PARALLELER HARDWARE 57
6.1 Einführung in OpenCL 57
6.2 Plattformmodell 58
6.3 Ausführungsmodell 59
6.4 Speichermodell 62
7 PARALLELISIERUNG DES RETE-ALGORITHMUS FÜR MASSIV PARALLELE HARDWARE 64
7.1 Bestimmung der Parallelitätsgranularität 64
7.2 Alpha-Matching 66
7.3 Beta-Matching 67
7.4 Node-Level-Parallelisierung 69
7.5 Regelauswertung 70
7.6 Zusammenfassung 72
8 PARTITIONIERUNG UND VERTEILUNG DER ARBEITSLAST 74
8.1 Alpha-Matching 74
8.2 Beta-Matching und Regelableitung 75
8.3 Verteilung und Parallelisierung 77
9 DATENSTRUKTUREN UND METHODEN FÜR EINE RESSOURCENSCHONENDE AUSFÜHRUNG 79
9.1 Dictionary-Encoding 79
9.1.1 Benötigte Daten und Aufbau einer naiven Implementierung . 80
9.1.2 Reduzierung der Hauptspeicherbelastung 81
9.2 Working-Memories 83
9.3 Tripel-Index-Struktur zur Erkennung von Duplikaten 85
9.3.1 Aufbau einer naiven Implementierung 85
9.3.2 Komprimierte Speicherung von Tripeln 86
9.3.3 Zusammenfasung 89
9.4 Codegenerierung für die GPU 90
10 EVALUATION 96
10.1 Testumgebung 96
10.1.1 Implementierung 96
10.1.2 Verwendete Datensätze 102
10.1.3 Verwendete Regelsätze 104
10.1.4 Hardwareumgebung 105

10.1.5 Bestimmung des notwendigen Speicherplatzes einer naiven Implementierung . . . 106
10.2 Versuchsdurchführung . . . 107
10.2.1 Dictionary-Encoding unter beschränkten Ressourcen . . . 108
10.2.2 Speicherverbrauch für das Reasoning . . . 110
10.2.3 Effektivität der Parallelisierung . . . 112
10.2.4 Large-scale Reasoning auf einzelnen Rechnern . . . 116
10.3 Vergleich zu verwandten Arbeiten . . . 119
10.4 Ergebnisauswertung . . . 122

III ZUSAMMENFASSUNG UND AUSBLICK 125
11 ZUSAMMENFASSUNG 126
12 FAZIT UND AUSBLICK 130

LITERATURVERZEICHNIS 132

ABBILDUNGSVERZEICHNIS

Abbildung 1 Grafische Darstellung des RDF-Graphs 14
Abbildung 2 Linking Open Data Cloud Diagram 2014 19
Abbildung 3 RETE-Netz der Regeln R_1 und R_2 41
Abbildung 4 RETE-Netz mit Working-Memories 43
Abbildung 5 Verschiedene Möglichkeiten der Netzwerk-Bildung 45
Abbildung 6 RETE-Optimierung: Most specific first 46
Abbildung 7 RETE-Optimierung: Most volatile condition last 47
Abbildung 8 RETE-Optimierung: Share join clusters among rules. 48
Abbildung 9 DADO Prozessorarchitektur 51
Abbildung 10 Abstrakte OpenCL Device-Architektur 59
Abbildung 11 Beispiel eines zweidimensionalen OpenCL Indexraums . . . 61
Abbildung 12 Mapping des OpenCL Speichermodells auf die Architektur einer AMD Readon 7970 GPU 63
Abbildung 13 Parallelisierung des Indexraums der Berechnungen des Alpha-Matching 67
Abbildung 14 RETE-Netzwerk während zwei aufeinanderfolgenden Iterationen des RETE-Algorithmus 68
Abbildung 15 Parallelisierung des Beta-Matching für einen Beta-Knoten . 69
Abbildung 16 RETE-Netzwerk der Regeln R_3 und R_4 70
Abbildung 17 Ableiten neuer Fakten auf massiv paralleler Hardware . . . 71
Abbildung 18 Partitionierung der Arbeitslast für das Alpha-Matching . . . 75
Abbildung 19 Partitionierung der Arbeitslast für das Beta-Matching 76
Abbildung 20 Node-Level-Parallelisierung 77
Abbildung 21 Datenstrukturen für das Dictionary-Encoding 81
Abbildung 22 Datenverteilung beim Dictionary-Encoding unter Verwendung des Festplattenspeichers 82
Abbildung 23 Unterschiedliche Aufbau-Möglichkeiten der Working-Memories für die RETE-Knoten der Regeln R_1 und R_2 84
Abbildung 24 RETE-Netzwerk zur Veranschaulichung der Host-Level-Parallelisierung 98
Abbildung 25 Architekturkonzept der Reasoner-Implementierung zur Umsetzung der Host-Level-Parallelisierung 99
Abbildung 26 Vergleich des pD* Reasonings unter Verwendung der verschiedenen Ausführungsstrategien 114
Abbildung 27 Vergleich des RDFS Reasonings unter Verwendung der verschiedenen Parallelisierungskonzepte 115

Abbildung 28 Darstellung der Reasoning-Zeit, verteilt auf das Alpha-Matching, Beta-Matching und Feuern von Regeln 118
Abbildung 29 Auslastung der GPU(s) während der einzelnen Reasoning-Schritte auf dem MacBook und auf der Workstation 119

TABELLENVERZEICHNIS

Tabelle 1 RDF/RDFS Ableitungsregeln 104
Tabelle 2 pD* Ableitungsregeln 105
Tabelle 3 Speicherverbrauch der notwendigen Datenstrukturen für eine naive Reasoner-Implementierung 107
Tabelle 4 Konfigurationsparameter der verwendeten Hardwareumgebungen 107
Tabelle 5 Ergebnisse des Dictionary-Encodings 109
Tabelle 6 Ergebnisse der Komprimierung der verwendeten Datensätze durch das Dictionary-Encoding 110
Tabelle 7 Anzahl abgeleiteter Fakten 111
Tabelle 8 Speicherverbrauch für die Tripel-Index-Struktur nach Anwendung der ρdf Regeln 111
Tabelle 9 Ausführungszeit in Sekunden für das pD* Reasoning unter Verwendung der verschiedenen Ausführungsstrategien . . . 113
Tabelle 10 Ausführungszeit in Sekunden für das RDFS Reasoning unter Verwendung der verschiedenen Parallelisierungskonzepte 115
Tabelle 11 Ausführungszeiten für das ρdf und RDFS Reasoning auf dem MacBook 117
Tabelle 12 Ausführungszeiten für das ρdf und RDFS Reasoning auf der Workstation 118

Tabelle 13 Vergleich der Ausführungszeiten der von Heino et al. vorgestellten Arbeit und der in dieser Arbeit eingeführten Implementierung . . . 121

LISTINGS

Listing 1 Beispielhafter Aufbau eines Dokuments zur Beschreibung von Informationen über Professoren unter Verwendung von Meta-Daten . . . 12
Listing 2 Beispiel-Faktenbasis im N3 Format . . . 13
Listing 3 Ausschnitt der RDFS-Ontologie zum zuvor eingeführten Beispiel . . . 15
Listing 4 Definition der Regel-Syntax . . . 39
Listing 5 Generierter Programmcode für das Alpha-Matching einer Regelbasis mit insgesamt fünf unterschiedlichen Alpha-Pattern 94

LISTE DER ALGORITHMEN

1 Berechnung der Werte v_l und v_r . . . 88
2 Vereinfachter Algorithmus für das parallele Alpha-Matching . . . 92
3 Algorithmus für das Alpha-Matching auf der GPU basierend auf den Eingabe-Regeln (generiert) . . . 93

1 EINLEITUNG

Semantische Technologien sind längst nicht mehr nur Gegenstand der Forschung, sondern kommen in verschiedenen Anwendungsbereichen zum Einsatz. Diese können sich von medizinischen und industriellen Anwendungsfällen bis hin zu Lösungen in der Finanz-, Medien- und Unterhaltungsindustrie erstrecken [AH08] [CHL07] [Gre11] [GDBG05] [KSR+09] [MRS09] [PBS12a] [PBS12b]. Als der wohl größte Anwendungsbereich wird das Semantische Web gesehen, das eine Erweiterung zum World Wide Web darstellt. Diese Erweiterung basiert darauf, dass Daten in einer maschinenlesbaren Form gespeichert werden und so die Möglichkeit zur Verknüpfung und Auswertung von Informationen durch Maschinen besteht. Aufbauend auf diesen Möglichkeiten lassen sich beispielsweise Ergebnisse von Suchanfragen durch die Berücksichtigung von Querbeziehungen oder Begriffsanalogien optimieren und Daten unterschiedlicher Herkunft zusammenführen und weiterverarbeiten [AH08] [BHL+14].

Als grundlegendes Datenformat semantischer Anwendungen dient das vom World Wide Web Consortium (W3C) konzipierte Resource Description Framework (RDF) [RDF14], dass die Formulierung logischer Aussagen über Ressourcen erlaubt. Eine mittels RDF beschriebene Aussage besteht aus den drei Einheiten *Subjekt*, *Prädikat* und *Objekt*, die gemeinsam ein *Tripel* bilden. Eine Vielzahl solcher Aussagen führt zu einer Wissensbasis, die nicht nur aus den explizit formulierten Informationen besteht, sondern auch implizit gegebene Fakten enthalten kann. Diese können durch logisches Schließen *materialisiert* werden, also als abgeleitetes Wissen der Faktenbasis explizit hinzugefügt werden, so dass sie für eine Weiterverarbeitung auch in einer expliziten Form zur Verfügung stehen.

Die während der Materialisierung anzuwendende Semantik bei dem im allgemeinen als *Reasoning* bezeichneten Prozess des Ableitens implizit gegebener Fakten kann sich in Abhängigkeit der verwendeten Inferenzregeln unterscheiden. Die *Inferenzregeln* beschreiben z.B. in Abhängigkeit der zugrunde liegenden Ontologiesprache, aus welchen expliziten Informationen Rückschlüsse auf implizite Fakten gezogen werden können. Eine *Ontolgiesprache* wiederum definiert den notwendigen Formalismus bestehend aus einer eindeutigen Spezifikation von Syntax und Semantik, um eine maschinenlesbare Repräsentation von Wissen zu ermöglichen [Fur14].

Das Ableiten der implizit gegebenen Informationen kann unterschiedliche Zwecke erfüllen. Einerseits lässt es eine Überprüfung auf Inkonsistenzen (widersprüchliche Aussagen) und eine Klassifizierung von Informationen zu, andererseits kann

es insbesondere Suchanfragen um zusätzliche Informationen anreichern bzw. die Suchergebnisse vervollständigen [Fur14] [Obe14].

1.1 PROBLEMBESCHREIBUNG

Im World Wide Web sind bereits eine Vielzahl verschiedener Datenquellen verfügbar, die frei verwendbare Informationen unter Nutzung von RDF anbieten. Ein Bestreben, aus dieser Datenvielfalt durch Verknüpfung der einzelnen Inhalte die Idee des semantischen Webs voranzutreiben, wurde bereits 2006 von Tim Berners-Lee skizziert [Ber06] und führte zur Prägung des Begriffs *Linked Data*. Er basiert auf der Annahme, dass das semantische Web erst dann von tatsächlichem Nutzen ist, wenn Informationen verschiedener Datenquellen verknüpft werden, so dass eine Maschine oder ein Mensch von einer Datenquelle zu relevanten Informationen einer weiteren Datenquelle gelangen kann. Ausgehend von dieser Idee entstand das Linked Open Data Cloud Projekt [BHu07] (LOD-Cloud), das Datenquellen in RDF veröffentlicht und Verknüpfungen (ebenfalls unter Verwendung von RDF) zwischen verschiedenen Datenquellen bereitstellt.

Während die tatsächliche Größe des semantischen Webs nicht beziffert werden kann, lässt sich anhand der Größe der LOD-Cloud bereits ein guter Eindruck von dem möglichen Umfang verfügbarer semantischer Informationen vermitteln. Die LOD-Cloud umfasste bereits im September 2011 295 Datensätze aus verschiedenen Domänen mit insgesamt rund 31,63 Milliarden Tripeln [Jen11]. Eine aktuellere Statistik über die LOD-Cloud wurde im April 2014 in [SBP14] veröffentlicht und zeigt eine Verdoppelung der Anzahl der Datensätze seit 2011. Eine weitere Quelle für öffentlich zugängliche RDF-Datensätze ist das Open Source Projekt Bio2RDF[1], das verschiedene biomedizinische Datensätze sowie Verknüpfungen zwischen diesen im RDF Format bereitstellt. Die Größe der dort verfügbaren Datensätze variiert zwischen einigen hunderttausend und rund fünf Milliarden Tripeln[2]. Der Umfang dieser Datenquellen verdeutlicht im Hinblick auf das semantische Web, aber auch im Hinblick auf die Verarbeitung einzelner Datensätze für dedizierte Anwendungen, die aufgrund der Größe entstehenden Herausforderungen für das Reasoning, die nur mit Konzepten der Lastverteilung und damit einhergehenden Parallelisierung bewältigt werden können [LZK+09]. Insbesondere vor dem Hintergrund schneller Antwortzeiten und performance-kritischer Anwendungen kann das Reasoning aber auch bei weitaus kleineren Datensätzen bereits zum „Flaschenhals" der Verarbeitung werden, wie beispielsweise in [PBSZ13a] gezeigt wurde.

1 Das Projekt ist unter www.bio2rdf.org zu erreichen. Die hier wiedergegebenen Informationen beziehen sich auf den Stand vom 05.09.2014

2 Stand vom 05.04.2015

Für die Realisierung klassischer Reasoner wird bisher kaum eine Form der Parallelisierung verwendet. Bei skalierbaren Systemen, die auf die Verarbeitung großer Datenmengen ausgelegt sind, findet diese hingegen hauptsächlich durch die Verwendung von mehreren Rechnern in einem Cluster Ausdruck. Des Weiteren sind diese Systeme häufig für einen speziellen Regelsatz bzw. für eine spezielle Ontologiesprache optimiert, so dass beispielsweise Inferenz-Regeln nicht als Regeln im klassischen Sinne zur Ausführungszeit geladen werden, sondern als Implementierungs-Vorschrift für Methoden dienen, die in einer aufeinander abgestimmten Reihenfolge ausgeführt werden und so zum gewünschten Ergebnis führen [KMK08] [LQWY11] [HP12] [UKM+12]. Somit konzentriert sich die Forschung im Bereich der leistungsfähigen Reasoner weitestgehend auf eine Parallelisierung und Lastverteilung über mehrere Rechner sowie auf die Implementierung einer gegebenen Semantik. Dadurch besteht vor allem bei der Verarbeitung großer Datensätze die Einschränkung, dass die beim Reasoning angewandte Semantik von der konkreten Reasoner-Implementierung abhängig ist und nicht durch domänenspezifische Regeln und zusätzliches Wissen erweitert werden kann [TAFK12].

Aufgrund der stetig wachsenden Menge semantischer Daten und einer weiten Verbreitung von Anwendungsfällen, die häufig kurze Reaktionszeiten voraussetzen, ergibt sich die Notwendigkeit der Entwicklung performanter und skalierbarer Reasoner unter Berücksichtigung des Ressourcenverbrauchs. Während durch den Einsatz von Clustern, bestehend aus einer Vielzahl an Rechnern, eine hohe Parallelität bei einer gleichzeitigen Steigerung der verfügbaren Rechenkapazität erreicht wird, werden Methoden zur effizienteren Nutzung der Ressourcen einzelner Rechner bisher kaum betrachtet (ausgenommen hiervon sind Arbeiten, die sich auf die Implementierung von Reasonern für ressourcenbeschränkte Geräte wie Smartphones und Sensornetzwerke konzentrieren, vgl. [SS11b] [TBKO09] [TKO11]). Diese Vorgehensweise hat sich bezüglich der verarbeitbaren Größe von Datensätzen und des erzielten Durchsatzes als zielführend erwiesen (vgl. [UKM+10] [ZQL+12] [LQWY12]), resultiert aber in einem großen Aufwand und hohen Kosten für die Verarbeitung großer Datensätze mit mehreren hundert Millionen Statements, wie sie beispielsweise in den zuvor genannten Quellen wie dem Bio2RDF Portal zur Verfügung stehen.

Neben der Betrachtung der Ressourceneffizienz findet auch der Aspekt der tatsächlichen Nutzung von Regeln zur Beschreibung von Wissen und Ableitungsregeln bei Systemen, die für große Datenmengen geeignet sind, bisher kaum Berücksichtigung. Ein Grund hierfür ist einerseits z.B. das genutzte Programmiermodell (z.B. MapReduce [DG04]), das eine Überführung einzelner Regeln in entsprechende Verarbeitungsschritte ermöglicht. Andererseits erlaubt die gezielte Implementierung von Ableitungsregeln weitreichende Optimierungen, die auf Eigenschaf-

ten der Semantik basieren und beispielsweise durch die Ordnung der Ausführungsschritte die Anzahl der notwendigen Operationen reduzieren können.

1.2 ZIELSETZUNG UND FORSCHUNGSFRAGEN

Aus den zuvor genannten Aspekten der Verarbeitung großer Datensätze unter Berücksichtigung des notwendigen Hardware-Einsatzes leitet sich die Forschungsfrage dieser Arbeit ab:

> Inwiefern lässt sich der Reasoning-Prozess parallelisieren, so dass er einerseits von der Leistungsfähigkeit massiv paralleler Hardware profitieren kann und andererseits unter den beschränkten Ressourcen eines einzelnen Rechners ein regelbasiertes Reasoning auf mehreren Milliarden Fakten erlaubt?

Die zuvor aufgeführte Forschungsfrage lässt sich in drei wesentliche Themenschwerpunkte untergliedern, die zu den folgenden Detail-Fragen führen:

- Wie lässt sich die zum Teil massiv parallele Architektur moderner Mehrkernprozessoren (CPU, GPU) für einen effizienten Reasoning-Prozess nutzen?
- Wie kann dieser Reasoning-Prozess so gestaltet werden, dass das resultierende System nicht nur eine bestimmte Semantik implementiert, sondern die Semantik in Form von Regeln definiert werden kann?
- Wie kann der Ressourcenbedarf für diesen Reasoning-Prozess soweit reduziert werden, dass eine Verarbeitung von Datensätzen mit mehreren Milliarden Fakten auch auf einzelnen Rechnern möglich ist?

Entsprechend der formulierten Forschungsfragen besteht die Zielsetzung der Arbeit darin, eine prototypische Reasoner-Architektur zu entwickeln, die unter Nutzung massiv paralleler Hardware auch für große Datenmengen eine schnelle Verarbeitung der Daten unter Anwendung von Regeln bietet. Aufgrund der Ausrichtung auf eine Verarbeitung von Daten auf einem einzelnen Rechner ohne Cluster-Betrieb werden keine „Web-Scale“-Datenmengen adressiert, also Datenmengen, die mit der Größe des semantischen Webs verglichen werden können. „Large-Scale“-Datensätze, die in dieser Arbeit als Datensätze mit einer Größe von vielen Millionen bis hin zu einigen Milliarden Fakten verstanden werden, sollen hingegen unterstützt werden. Demnach soll unter Reduzierung der eingesetzten Ressourcen (Grafikkarte(n) statt Cluster) ein paralleles und regelbasiertes System zum Ableiten von implizit gegebenen Fakten aus RDF-Daten konzipiert und erprobt werden.

1.3 FORSCHUNGSGEGENSTAND

Die Abfrage semantischer Daten geschieht häufig unter Verwendung von Anfragesprachen wie SPARQL [PS08]. Dabei unterliegt SPARQL für gewöhnlich der *Closed World Assumption*, was bedeutet, dass Ergebnisse von Suchanfragen nur solche Informationen beinhalten, die explizit gegeben sind. Um die Vollständigkeit von Anfrageergebnissen gewährleisten zu können, muss bereits vor oder während der Suchanfrage eine für den betroffenen Teilausschnitt vollständige Faktenbasis materialisiert werden. Im Falle der Nutzung eines regelbasierten Ansatzes werden die unterschiedlichen Vorgehensweisen als *Forward Chaining* (vollständige Materialisierung der Faktenbasis vor Ausführung einer Anfrage) und *Backward Chaining* (nur der für die Anfrage notwendige Teilausschnitt wird materialisiert) bezeichnet [RLB+90]. Beide Vorgehensweisen haben Vor- und Nachteile die sich je nach Anwendungsfall unterschiedlich auswirken können. Forward Chaining erfordert insbesondere initial einen großen Aufwand zur vollständigen Materialisierung aller implizit gegebenen Fakten. Der Vorteil besteht darin, dass zur Beantwortung von Anfragen kein weiterer Reasoning-Prozess notwendig ist, solange sich die Faktenbasis nicht ändert. Beim Backward Chaining wird hingegen bei jeder Anfrage genau der Teilausschnitt materialisiert, der zur Beantwortung der Anfrage beiträgt. Während sich diese Vorgehensweise grundsätzlich für sich stetig ändernde Faktenbasen anbietet, bedeutet die anfragenbezogene Materialisierung insbesondere bei großen Datenmengen häufig eine langsame Reaktionszeit.

Ausgehend von der Fokussierung großer Datenmengen und unter Betrachtung der zuvor aufgeführten Datensätze bestehend aus mehreren hundert Millionen statischer Fakten, wird zur Erreichung der Zielsetzung dieser Arbeit entsprechend der aufgeführten Argumentation zu den Vorgehensweisen des Reasonings die Verwendung von Forward-Chaining-Reasoning verfolgt. Ein besonderer Fokus liegt dabei auf der Algorithmik zur Parallelisierung des Reasoning-Prozesses, also dem Ableiten von Fakten aus einer gegebenen Faktenbasis. Die so entstehende Faktenbasis kann anschließend (ggf. nach vorheriger Indexierung) von entsprechenden Query-Engines, wie sie für SPARQL existieren, genutzt werden.

Um die formulierte Zielsetzung zu erreichen, ist zudem die Verwendung eines Ansatzes notwendig, der die Definition der anzuwendenden Semantik durch Regeln ermöglicht. Zu den Vorteilen bei der Verwendung eines regelbasierten Systems zählen zum einen, dass ohne zusätzlichen Implementierungsaufwand unterschiedliche Semantik genutzt werden kann, zum anderen sind auch von (quasi) standardisierten Ontologiesprachen abweichende Regeln leicht zu realisieren. Ein Nachteil besteht jedoch in der Tatsache, dass sich komplexe Ontologiesprachen nicht immer vollständig durch Regeln ausdrücken lassen bzw. die Logik für ein regelbasiertes System nicht immer entscheidbar ist. Diese Einschränkung wird je-

doch im Hinblick auf die insbesondere bei der Verarbeitung großer Datenmengen weit verbreitete Nutzung der monotonen Regeln der RDFS sowie der pD* Semantik [tH05] (auch OWL-Horst genannt), die sich jeweils durch ein entscheidbares Regelwerk ausdrücken lassen, in dieser Arbeit nicht weiter berücksichtigt und als Teil einer Fortführung der Forschungsarbeiten gesehen. Entsprechend sollen nur monotone Regelformulierungen unterstützt werden, so dass die anzuwendende Semantik immer entscheidbar ist und während des Reasonings ausschließlich neue Fakten abgeleitet werden, ohne die Gültigkeit bestehender Fakten durch Negationen zu widerrufen [HKRS08].

Neben der Erforschung des regelbasierten Reasonings durch die Nutzung massiv paralleler Hardware findet zusätzlich eine Betrachtung des notwendigen Ressourcenbedarfs während der Ausführung statt. Ein häufig limitierender Faktor bei der Verarbeitung großer Datenmengen, insbesondere bei der Nutzung von regelbasierten Systemen, stellt der verfügbare Speicher dar [Mad03]. Zur Erreichung der Zielsetzung der Verarbeitung von Large-scale-Datensätzen auf einzelnen Rechnern werden die ressourcenintensiven Schritte identifiziert und durch Maßnahmen der Strukturierung, Auslagerung und Komprimierung von Daten insbesondere in Bezug auf den Speicherbedarf optimiert.

Zusammenfassend konzentriert sich die vorliegende Arbeit auf die Erforschung eines hochgradig parallelisierten und regelbasierten Ansatzes zum Ableiten von RDF-Fakten unter Anwendung von Forward Chaining für monotone Regelformulierungen. Als wesentlicher Forschungsgegenstand wird dabei einerseits die Übertragung und Adaption von Algorithmen aus dem Bereich der Wissensgenerierung auf die massiv parallele Architektur moderner Grafikprozessoren angesehen und andererseits die Reduzierung des für die Algorithmik notwendigen Ressourcenbedarfs.

1.4 AUFBAU DER ARBEIT

Die Arbeit gliedert sich in drei Bereiche. Teil I gibt zunächst einen Überblick über die zugrunde liegenden Technologien und Ontologiesprachen des semantischen Webs. Darauf aufbauend werden die Grundlagen des Reasonings erläutert, um anschließend einen umfassenden Überblick über das parallele und das regelbasierte Reasoning geben zu können. Zur Einführung der regelbasierten Verarbeitung wird in Kapitel 4 zunächst detailliert auf Production Systems eingegangen, die das grundlegende Konzept der regelbasierten Verarbeitung beschreiben und somit die Ausgangsbasis für den RETE-Algorithmus[3] darstellen. Dieser wird anschließend im Detail betrachtet, da er als Grundlage für den in dieser Arbeit ent-

3 Der RETE-Algorithmus ist ein Algorithmus, der eine effiziente Regelverarbeitung auch bei großen Datenmengen erlaubt. Er kommt bei vielen Expertensystemen und Rule-Engines zum Einsatz.

wickelten Reasoning-Prozess dient. Kapitel 5 gibt anschließend einen Überblick über relevante Arbeiten im Bereich der Parallelisierung von Production Systems und beschreibt Konzepte, die auch im weiteren Verlauf der Arbeit zum Tragen kommen.

Teil II beschreibt zunächst die Grundlagen der parallelen Programmierung unter Verwendung von OpenCL[4], um in Kapitel 7 die theoretische Überführung des RETE-Algorithmus zur Ausführung auf einer massiv parallelen Hardware zu beschreiben. Die Inhalte dieses Kapitels wurden bereits teilweise in

> Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Rule-based Reasoning on Massively Parallel Hardware. In: 9th International Workshop on Scalable Semantic Web Knowledge Base Systems. (2013)

veröffentlicht. Kapitel 8 stellt aufbauend auf den Ergebnissen der Überführung des RETE-Algorithmus auf eine massiv parallele Ausführungsumgebung Konzepte zur Partitionierung und Verteilung der Arbeitslast vor, die bereits in

> Peters, M., Brink, C., Sachweh, S., Zündorf, A.: Scaling Parallel Rule-based Reasoning. In: 11th Extended Semantic Web Conference (ESWC). (2014)

diskutiert wurden. Kapitel 9 führt Konzepte für die ressourcenschonende Ausführung des Reasoning-Prozesses ein, um das Ziel der Verarbeitung möglichst großer Datensätze auf einzelnen Rechnern zu adressieren. Dabei werden sowohl vorverarbeitende Schritte wie das *Dictionary-Encoding* betrachtet als auch die für den Reasoning-Prozess notwendigen Datenstrukturen. Die in Kapitel 9 vorgestellten Konzepte finden sich zum Teil auch in der folgenden Veröffentlichung wieder:

> Peters, M., Sachweh, S., Zündorf, A.: Large scale rule-based Reasoning using a Laptop. In: 12th Extended Semantic Web Conference (ESWC). (2015)

In Kapitel 10 erfolgt eine Evaluation der erarbeiteten Konzepte als Gesamtsystem unter Verwendung einer prototypischen Implementierung in Java. Als Ausführungsumgebung werden einerseits ein Laptop und andererseits eine leistungsstärkere Workstation verwendet, um die Auswirkungen der unterschiedlichen Konzepte im Detail evaluieren zu können. Des Weiteren erfolgt in Kapitel 10 eine vergleichende Betrachtung mit verwandten Arbeiten. Die Ergebnisse des Kapitels wurden ebenfalls zum Teil in den bereits aufgeführten Veröffentlichungen diskutiert.

Der letzte Teil der Arbeit (Teil III) fasst die erreichten Ergebnisse zusammen, stellt sie der Zielsetzung der Arbeit gegenüber und zieht ein abschließendes Fazit. Im Ausblick werden vor dem Hintergrund der vorgestellten Konzepte aufbauende Fragestellungen aufgegriffen und Möglichkeiten aufgezeigt, die Forschung in diesem Bereich fortzuführen.

---

4 OpenCL ist ein offener Standard zur Programmierung heterogener Multi-Core Systeme.

Teil I

GRUNDLAGEN UND VERWANDTE ARBEITEN

2 SEMANTISCHES WEB

Im Folgenden wird ein Überblick über die Idee des semantischen Webs und der zugrundeliegenden Technologien, Standards und Frameworks gegeben. Das Kapitel dient somit der Einführung von Grundlagen, die im weiteren Verlauf der Arbeit aufgegriffen werden. Aufgrund der Komplexität des Themenfeldes und der sich daraus ableitenden Vielzahl an Standardisierungsbemühungen und -vorschlägen erhebt das Kapitel keinen Anspruch auf Vollständigkeit, sondern fokussiert die für diese Arbeit relevanten Aspekte und Frameworks. Ein umfassender Überblick kann beispielsweise in [KBM08] gefunden werden.

2.1 EINFÜHRUNG IN DAS SEMANTISCHE WEB

Der Begriff des *semantischen Webs* wurde 2001 durch Tim Berners-Lee, dem Begründer des World Wild Web und Direktor des World Wide Web Consortium (W3C), James Hendler und Ora Lassila geprägt, die die grundlegende Idee wie folgt beschreiben:

> *„The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* [BLHL01]

Demnach existiert das semantische Web nicht parallel zum World Wide Web, sondern stellt eine Erweiterung dar, in der Informationen mit einer wohldefinierten Bedeutung assoziiert sind, die es Menschen und Computern ermöglichen, diese zu verstehen[1] und zu nutzen. Im Gegensatz zum klassischen World Wide Web, in dem Informationen sowie deren Bedeutung und inhaltliche Relevanz weitestgehend nur von Menschen verstanden werden können, sollen somit auch Maschinen in die Lage versetzt werden, diese gewinnbringend und für den Menschen zielführend zu verarbeiten.

Eine klassische HTML-Seite beispielsweise kann Informationen über die Sprechzeiten eines Dozenten enthalten. Die HTML-Seite hat einen wohlgeformten Aufbau und besteht aus Elementen, die den einzelnen Textteilen eine Struktur verleihen und dem Computer eine für den Menschen gut verständliche Darstellung

1 Im Falle von Computern sollen die Informationen nicht tatsächlich verstanden werden, sondern wie bereits in [HKRS08] dargelegt, vielmehr durch Maschinen auf eine für den Menschen sinnvolle und zielführende Art und Weise genutzt werden können. Diese Form der Interpretation wird auch im Folgenden zugrunde gelegt.

der Informationen ermöglichen. Trotz der Verwendung von HTML ist der Computer jedoch nicht in der Lage, die Informationen auch zu verstehen bzw. deren Bedeutung zu erfassen. Eine Websuche nach „hat Professor Bob heute Sprechstunde“ könnte also die tatsächlichen Sprechzeiten, auch wenn sie in der HTML-Seite enthalten sind, nicht als konkrete Antwort liefern. Stattdessen würde bestenfalls eine Seite gefunden werden, die inhaltlich sowohl die Begriffe „Professor Bob“ und „Sprechstunde“ beinhaltet. Ob die gelieferten Informationen tatsächlich eine Antwort auf die gestellte Suchanfrage liefern, kann dabei nicht berücksichtigt werden.

Durch die Idee des semantischen Webs, also der Bereitstellung von Informationen in einem maschinenlesbaren Format, ließe sich das Suchergebnis dahingehend optimieren, dass dem Benutzer nicht nur Verweise auf Internetseiten präsentiert werden, sondern z.B. die gewünschte Antwort in aufbereiteter Form und unter Berücksichtigung der Zeitangabe „heute“. Zu den wichtigsten Technologien, um diese Vision zu verwirklichen, gehören Meta-Daten, Ontologien sowie Logik und Inferenz [BACT07]:

Meta-Daten: Meta-Daten sind Daten über Daten und dienen in verschiedenen Anwendungskontexten der Beschreibung von Informationsressourcen, um diese besser auffindbar zu machen und Beziehungen zwischen Ressourcen auch in einer heterogenen Landschaft herzustellen [PB06]. Letztendlich beschreiben Meta-Daten, wie Daten verstanden und genutzt werden können. Dazu bedarf es einer standardisierten bzw. einheitlichen Notation und Verwendung von Meta-Daten, zu deren Zweck im semantischen Web sich das ursprünglich vom W3C als Standard zur Beschreibung von Meta-Daten konzipierte RDF etabliert hat.

Ontologien: Der ursprünglich aus der Philosophie stammende Begriff der Ontologie wird in verschiedenen Disziplinen unterschiedlich aufgefasst und interpretiert. Im Bereich der semantischen Technologien und der Wissensverarbeitung ist eine der meist verwendeten Definitionen die von Thomas R. Gruber, der Ontologien als *„explicit specification of a conceptualization“* [Gru93] beschreibt. Diese Definition wurde von Willem Nico Borst ergänzt, der eine Ontologie wie folgt definiert *„formal specification of a shared conceptualization“* [Bor97]. Damit ergänzt er die ursprünglich von Gruber aufgestellte Definition um die Notwendigkeit, dass es sich bei dem Verständnis für die conceptualization nicht um eine rein individuelle Sicht handelt, sondern um eine im Allgemeinen so verstandene Sicht. Die *conceptualization* wird dabei als abstrakte, vereinfachte Sicht auf die Welt oder die Domäne, die beschrieben werden soll, verstanden. Zum anderen wird die Definition um den Aspekt der formalen Beschreibung ergänzt, die letztendlich die Verarbeitung durch Maschinen ermöglicht [SS09]. Basierend auf diesem Verständnis wurden bei-

de Definitionen von Studer et al. als *„An ontology is a formal, explicit specification of a shared conceptualisation"* [SBF98] zusammengefasst.

In Anlehnung an die zuvor aufgeführten Definitionen wird in dieser Arbeit der Begriff „Ontologie" wie folgt aufgefasst:

> *Eine Ontologie besteht aus einem Netz von Informationen, die über logische Beziehungen miteinander verknüpft sein können. Dem Aufbau der Wissensbasis (äquivalent zu Ontologie) liegt dabei eine formale Beschreibung zugrunde, die die Basis für die maschinelle Verarbeitung bildet.*

Des Weiteren wird zwischen Schema- (TBox) und Instanz- (ABox) Informationen unterschieden, die auch als Konzepte einer Domäne und das Wissen über die Entitäten oder Instanzen der Konzepte und deren Beziehungen untereinander bezeichnet werden können.

Logik und Inferenz: Beschreibungslogiken (Description Logics oder DLs) sind eine Familie von Sprachen zur Wissensrepräsentation [SS09]. Sie erlauben auf eine strukturierte und formale Weise das Konstruieren einer Wissensbasis, auf deren Grundlage Inferenzregeln angewandt werden können. Durch das Anwenden von Inferenzregeln können aus bekannten wahren Aussagen neue wahre Aussagen abgeleitet werden. Somit besitzt Beschreibungslogik einerseits eine wohldefinierte Semantik und erlaubt andererseits über eine Wissensbasis zu schließen, also aus vorhandenem Wissen Neues zu gewinnen. Aufgrund dieser Eigenschaften eignen sich Beschreibungslogiken zur Definition von Ontologiesprachen, was sich z.B. in der Verwendung einer Beschreibungslogik für die vom W3C entwickelten Ontologiesprache OWL [MvH04] zeigt.

Das Gesamtkonzept des semantischen Webs bzw. einer semantischen Anwendung ergibt sich aus einem Zusammenspiel der zuvor aufgeführten Technologien. Basierend auf einer Beschreibungslogik können Ontologiesprachen entwickelt werden, die durch eine explizite Angabe von Meta-Daten die Bedeutung von Informationen auch für Maschinen verwendbar machen. Für das einführende Beispiel der Suche nach den Sprechzeiten eines Dozenten könnte z.B. eine Ontologie definiert werden, die eine Beschreibung der Sprechstunde wie in Listing 1 erlaubt und darauf aufbauend eine semantische Suche ermöglicht.

```
<university>
    <staff>
        <professors>
            <professor>
                <name>Bob</name>
                <office-hours>
                    <thursday>
                        <time>16:00-18:00</time>
                    </thursday>
                </office-hours>
                ...
            <professor>
        <professors>
        ...
    </staff>
    ...
</university>
```

Listing 1: Beispielhafter Aufbau eines Dokuments zur Beschreibung von Informationen über Professoren unter Verwendung von Meta-Daten (eingefasst in < >), die zur Beschreibung von Instanzdaten verwendet werden

2.2 RESOURCE DESCRIPTION FRAMEWORK (RDF)

Das Resource Description Framework [RDF14] ist eine formale Sprache, die sich zur Beschreibung von Ressourcen im semantischen Web etabliert hat [LQWY12]. Das ursprüngliche Ziel lag in der Darstellung von Meta-Daten über Web Ressourcen, z.B. zur Angabe von Autorinnen und Autoren oder Lizenzinformationen von Webseiten [HKRS08]. Später erweiterte sich der Anwendungsbereich auf die allgemeine Darstellung semantischer Informationen, wodurch sich RDF zum grundlegenden Darstellungsformat auch für komplexe Ontologiesprachen entwickelt hat.

RDF kann zur Beschreibung beliebiger Ressourcen im Web genutzt werden. Zur Identifizierung und eindeutigen Referenzierung von Ressourcen werden Unified Resource Identifier (URIs) verwendet, die durch weitere Eigenschaften (Properties) beschrieben werden können [Den12]. Durch die Beschreibung von Ressourcen durch Eigenschaften mit Werten können Aussagen gebildet werden, die jeweils aus einem *Subjekt*, *Prädikat* und *Objekt* bestehen und so ein *Tripel* bilden. Werte können sowohl aus einer Referenz (URI) auf eine weitere Ressource bestehen als auch aus einfachen Datenwerten wie der Zahl 42 (*Literal*), deren Interpretation in Abhängigkeit des jeweiligen Datentyps erfolgen kann. Die Angabe des Datentyps

```
@prefix uni: <http://university.org/kassel#>

uni:Bob     uni:hasName "Bob"^^xsd:String ;
            uni:hasPosition uni:Professor ;
            uni:officeLocation uni:ParkStreet ;
            uni:officeHours uni:Monday ;
            uni:fieldOfResearch uni:SemanticWeb .
uni:Peter   uni:hasName "Peter"^^xsd:String;
            uni:hasPosition uni:ResearchAssistant ;
            uni:officeLocation uni:ParkStreet ;
            uni:worksFor uni:Bob ;
            uni:fieldOfResearch uni:SemanticWeb .
```

Listing 2: Beispiel-Faktenbasis im N3 Format

erfolgt durch eine Ergänzung des in Anführungszeichen eingefassten Datenwerts gefolgt durch eine mittels "^^" getrennte Datentyp-URI. Zur Gewährleistung einer einheitlichen Verwendung von Datentypen empfiehlt RDF die Verwendung der in XML Schema definierten Datentypen. Ohne Angabe eines Datentyps wird der jeweilige Wert als einfache Zeichenkette interpretiert.

Um eine einheitliche und maschinenlesbare Repräsentation von RDF-Statements zu gewährleisten, haben sich verschiedene Repräsentation wie RDF/XML [Dav04] oder Notation3 (N3) [Tim11] etabliert. Während RDF/XML verhältnismäßig viel Overhead bedingt, ist insbesondere N3 auf eine gute Lesbarkeit ausgelegt. Listing 2 zeigt eine Beispiel-Faktenbasis, die das bereits eingeführte Beispiel aus dem Universitätskontext näher beschreibt. Die Aussagen sind in N3-Notation angegeben, deren mit `http://university.org/kassel` beginnende URI durch die Nutzung des Präfix „uni:“ abgekürzt ist. Die Einführung des Präfixes erfolgt unter Verwendung von „@prefix“ und der Angabe einer Kurzschreibweise für eine URI. Alle Aussagen in N3 bestehen aus dem bekannten Aufbau *Subjekt*, *Prädikat*, *Objekt*, während ein Semikolon dazu genutzt werden kann, weitere Eigenschaften eines Subjekts zu beschreiben [SS09]. Abgeschlossen werden Aussagen durch Verwendung eines Punktes.

Die in der Faktenbasis aufgeführten Tripel beschreiben einen Graphen an Informationen, bestehend aus Knoten (Ressourcen) und gerichteten Kanten (Eigenschaften). In der grafischen Darstellung werden sowohl Knoten als auch Kanten mit den jeweils eindeutigen Bezeichnern beschriftet. Auf diese Weise ergibt sich für die Tripel aus Listing 2 der in Abbildung 1 gezeigte Graph. Die Darstellung von Literalen in der grafischen Repräsentation erfolgt durch eine rechteckige Umrandung, während URI-Bezeichner durch eine ovale Umrandung unterschieden werden. Literale können selber nicht durch weitere Eigenschaften beschrieben wer-

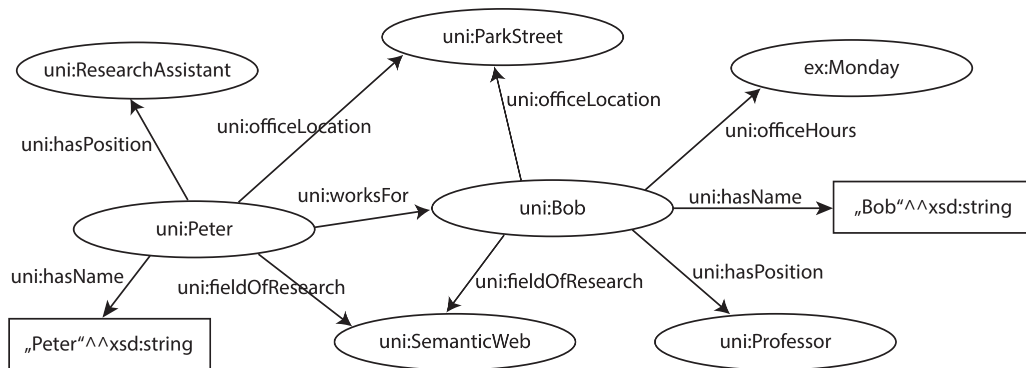


Abbildung 1: Grafische Darstellung des RDF-Graphs

den. Entsprechend kann z.B. der Name von *Peter* auch kein Ausgangspunkt für weitere Kanten in den Graphen sein, was gleichzeitig die Verwendung eines Literals als Subjekt in einem Tripel verbietet.

2.3 RDF-SCHEMA

Durch die alleinige Verwendung von RDF ist es nicht möglich, einen Rahmen bzw. ein Schema zur Beschreibung einer Gruppe von ähnlichen Objekten, beispielsweise aller Professorinnen und Professoren einer Hochschule, bereitzustellen. Zu diesem Zweck wurde die RDF Beschreibungssprache RDF-Schema (RDFS) [BG00] entwickelt, die eine Erweiterung des RDF-Vokabulars ist und Mechanismen zur Beschreibung von Typen oder Klassen von Ressourcen sowie deren Eigenschaften bereitstellt. Als praktischer Anwendungsfall ließe sich beispielsweise für das Konzept *Professor* die Semantik der Eigenschaft *fieldOfResearch* so definieren, dass durch diese Eigenschaft das Lehrgebiet einer Professorin oder eines Professors angegeben wird. Zusätzlich erlaubt RDFS die Definition der Domain und des Wertebereichs für eine Eigenschaft. Beispielsweise könnte für *fieldOfResearch* die Domain (rdfs:domain) *Professor* festgelegt werden, um nur Instanzen vom Typ Professor diese Eigenschaft zuweisen zu können. Der Wertebereich (rdfs:range) wiederum schränkt die möglichen referenzierbaren Instanzen ein, so dass die Eigenschaft *fieldOfResearch* beispielsweise nur auf Instanzen vom Typ *ResearchField* verweisen kann. Somit erlaubt RDFS die Definition eines möglichen Wertebereichs für Eigenschaften, die Bedeutung von Eigenschaften, welche Beziehungen zu anderen Eigenschaften bestehen und welche Arten von Ressourcen diese Eigenschaft nutzen dürfen. Mit diesen Möglichkeiten beschreibt RDFS noch keine konkrete Anwendungsdomäne, in der Klassen oder Eigenschaften definiert sind, sondern stellt vielmehr die Grundlage zur Entwicklung entsprechender Schemata zur Ver-

```
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
@prefix uni:    <http://university.org/kassel#>

uni:Staff               rdf:type        rdfs:Class .
uni:Professor           rdf:type        rdfs:Class ;
                        rdfs:subClassOf uni:Staff .
uni:ResearchAssistant   rdf:type        rdfs:Class ;
                        rdfs:subClassOf uni:Staff .

uni:worksFor            rdf:type        rdf:Property ;
                        rdfs:domain     uni:ResearchAssistant ;
                        rdfs:range      uni:Professor .

uni:Bob                 rdf:type        uni:Professor .
uni:Peter               rdf:type        uni:ResearchAssistent ;
                        uni:worksFor    uniBob .
```

Listing 3: Ausschnitt der RDFS-Ontologie zum zuvor eingeführten Beispiel

fügung. Diese Möglichkeit der Spezifikation von Schemawissen, deren Umfang in diesem Kapitel nur in Ansätzen beschrieben ist, macht RDFS zu einer Ontologiesprache (aufgrund der verhältnismäßig geringen Ausdrucksmächtigkeit eine sogenannte *leichtgewichtige* Ontologiesprache), die eine Beschreibung semantischer Abhängigkeiten innerhalb einer bestimmten Domain erlaubt [HKRS08].

Listing 3 zeigt einen Ausschnitt einer möglichen RDFS-Ontologie, die auf dem zuvor eingeführten Beispiel der Hochschule (Listing 2 und Abbildung 1) aufbaut. Zunächst werden die drei Klassen uni:Staff, uni:Professor und uni:ResearchAssistant eingeführt, wobei uni:Professor und uni:ResearchAssistant jeweils eine Subklasse von uni:Staff sind. In der weiteren Definition werden einige Meta-Properties von RDFS verwendet, um die Eigenschaft uni:worksFor näher zu beschreiben und sowohl die Domäne (uni:ResearchAssistant) als auch den Wertebereich (uni:Professor) zu definieren. Die so beschriebene Eigenschaft bildet gemeinsam mit den angegebenen Klassen die TBox der Ontologie und stellt somit die Schema-Informationen dar.

Die ABox der in Listing 3 angegebenen Ontologie besteht aus zwei Instanzen (uni:Bob und uni:Peter). Die Instanz Bob (uni:Bob) ist vom Typ Professor während Peter (uni:Peter) vom Typ ResearchAssistant ist. Zusätzlich wird Peter durch die Eigenschaft uni:worksFor beschrieben, die angibt, dass Peter für Bob arbeitet.

2.4 WEB-ONTOLOGIESPRACHEN

RDFS wurde bereits als leichtgewichtige Ontologiesprache eingeführt. Sie zeichnet sich durch eine einfache und beschränkte Semantik aus, was einerseits zu einer beherrschbaren Komplexität bei der Ausführung von Reasoning-Algorithmen führt, aber andererseits auch die Ausdrucksmächtigkeit einschränkt. Für Anwendungsfälle, die eine höhere Expressivität verlangen, wurde im Februar 2004 die Web Ontology Language (OWL) [MvH04] vom W3C standardisiert, die seit Oktober 2009 in der Version *OWL 2* vorliegt. Die Entwicklung von OWL 2 diente der Kompromissfindung zwischen Expressivität und dem effizienten Schlussfolgern, da eine hohe Ausdrucksstärke zur Bereitstellung von implizitem Wissen auch immer zu einer hohen Komplexität bis hin zur Unentscheidbarkeit beim Schließen von Wissen führen kann [HKRS08]. Aus diesem Grund teilt sich OWL 2 in drei Profile auf, die je nach Anwendungsfall genutzt werden können und sich in den Ausdrucksmöglichkeiten unterscheiden [W3C12]. OWL 2 EL eignet sich im Besonderen für ein effizientes Reasoning unter Verwendung einer Vielzahl an Schema-Informationen wie Klassen und Eigenschaften. Bei dem Profil OWL 2 QL hingegen liegt der Fokus auf einer schnellen Verarbeitung von Queries über einer großen Menge an Instanzinformationen. Das dritte Profil, OWL 2 RL, ist speziell für semantische Anwendungen gedacht, die hohe Anforderungen an die Skalierbarkeit stellen ohne dabei die Expressivität zu sehr einzuschränken. Ein weiteres Merkmal besteht darin, dass ein Reasoning System für OWL 2 RL unter Verwendung eines regelbasierten Ansatzes implementiert werden kann.

Eine weitere, weit verbreitete Ontologiesprache ist pD* [tH05]. Anders als OWL ist pD* keine Empfehlung des W3C, sondern eine Semantik, die von Herman J. ter Horst vorgestellt wurde und einige OWL-Fragmente mit RDFS kombiniert. pD* besitzt eine geringere Expressivität als OWL, da nur einige Sprachkonstrukte übernommen wurden, die im Verhältnis zum Nutzen eine relativ geringe Komplexität bei der Ausführung von Reasoning Algorithmen bedeuten. Des Weiteren kann die Semantik von pD* ebenso wie die von RDFS vollständig in Form von Ableitungsregeln ausgedrückt werden, was die Hürde der Implementierung reduziert. Aus diesem Grund kommen vor allem bei Anwendungsfällen mit großen Datensätzen, bei denen die Ausführungsgeschwindigkeit sowie Skalierbarkeit eine große Rolle spielen, in Abhängigkeit der notwendigen Ausdrucksmächtigkeit häufig RDFS oder pD* zum Einsatz (vgl. z.B. [GM10] [HP12] [KMK08] [LQWY11] [LQWY12] [UKM+12]).

2.5 LINKED DATA

Im Allgemeinen lässt sich *Linked Data* als die Veröffentlichung von strukturierten Daten beschreiben, die durch Verlinkungen an Nutzen gewinnt. Unter einer technischeren Betrachtungsweise ergänzt sich dieses Verständnis zu einer Veröffentlichung von Daten im Web in einer maschinenlesbaren Form mit einer explizit definierten Bedeutung, die Links zu anderen Datensätzen enthalten und entsprechend selber auch durch andere Datensätze verlinkt sein können [BHBL09]. Die Veröffentlichung und Verlinkung der Daten erfolgt unter Verwendung von RDF, wodurch die Heterogenität der jeweiligen Datenquellen verborgen wird. Durch diese Vorgehensweise soll ein Paradigmenwechsel vom „Web of Documents“ hin zum „Web of Data“ vollzogen werden [SP09].

Wie bereits in Kapitel 1 beschrieben, stammt die Idee zu Linked Data von Tim Berners-Lee [Ber06]. Um seiner Vision näher zu kommen beschrieb Tim Berners-Lee nicht nur seine Idee, sondern stellte auch vier Regeln auf, die bei der Veröffentlichung von Daten eingehalten werden sollten, um eine einheitliche Repräsentation und Verwendung sicherzustellen:

- Verwende URIs als Namen für Dinge.
- Verwende HTTP URIs, damit Leute diese Namen nachschlagen und referenzieren können.
- Wenn jemand den Namen nachschlägt, stelle nützliche Informationen unter Verwendung der Standards (RDF, SPARQL) bereit.
- Biete Links zu anderen URIs, so dass weitere Informationen entdeckt werden können.

Unter Berücksichtigung der oben aufgeführten Regeln wurde seit Bekanntwerden der Idee eine Vielzahl an Datensätzen veröffentlicht, die bereits verschiedene semantische Anwendungen und Dienste im Web ermöglichen. Viele der Datensätze sind in der Linked Open Data Cloud (LOD-Cloud) zusammengefasst [Jen11] [SBP14]. Die insgesamt über 1000 Datensätze [SBP14] stammen aus unterschiedlichen Domänen, die in Abbildung 2 in verschiedenen Farben dargestellt sind.

Die Kreisgröße eines Datensatzes in Abbildung 2 korreliert mit der jeweiligen Anzahl an Tripeln. Entsprechend variiert die Kreisgröße von *sehr groß* für Datensätze mit über einer Milliarden Fakten bis zu *sehr klein* für Datensätze mit weniger als 10.000 Fakten. Die Dicke der Pfeile hat eine ähnliche Bedeutung. Sie korreliert mit der Anzahl der Links, die zwischen zwei Datensätzen bestehen und visualisiert bei einer dünnen Darstellung zwischen 50 und 1.000 Links, während dicke Pfeile mehr als 100.000 Links bedeuten.

Die LOD-Cloud verdeutlicht die Vielfalt der als RDF verfügbaren Datenquellen, die für semantische Anwendungen genutzt werden können. Sie zeigt aber auch, welche Datenmengen ggf. verarbeitet werden müssen, um aus den vorhandenen Fakten einen Mehrwert zu generieren und z.B. optimierte Suchergebnisse bereitzustellen.

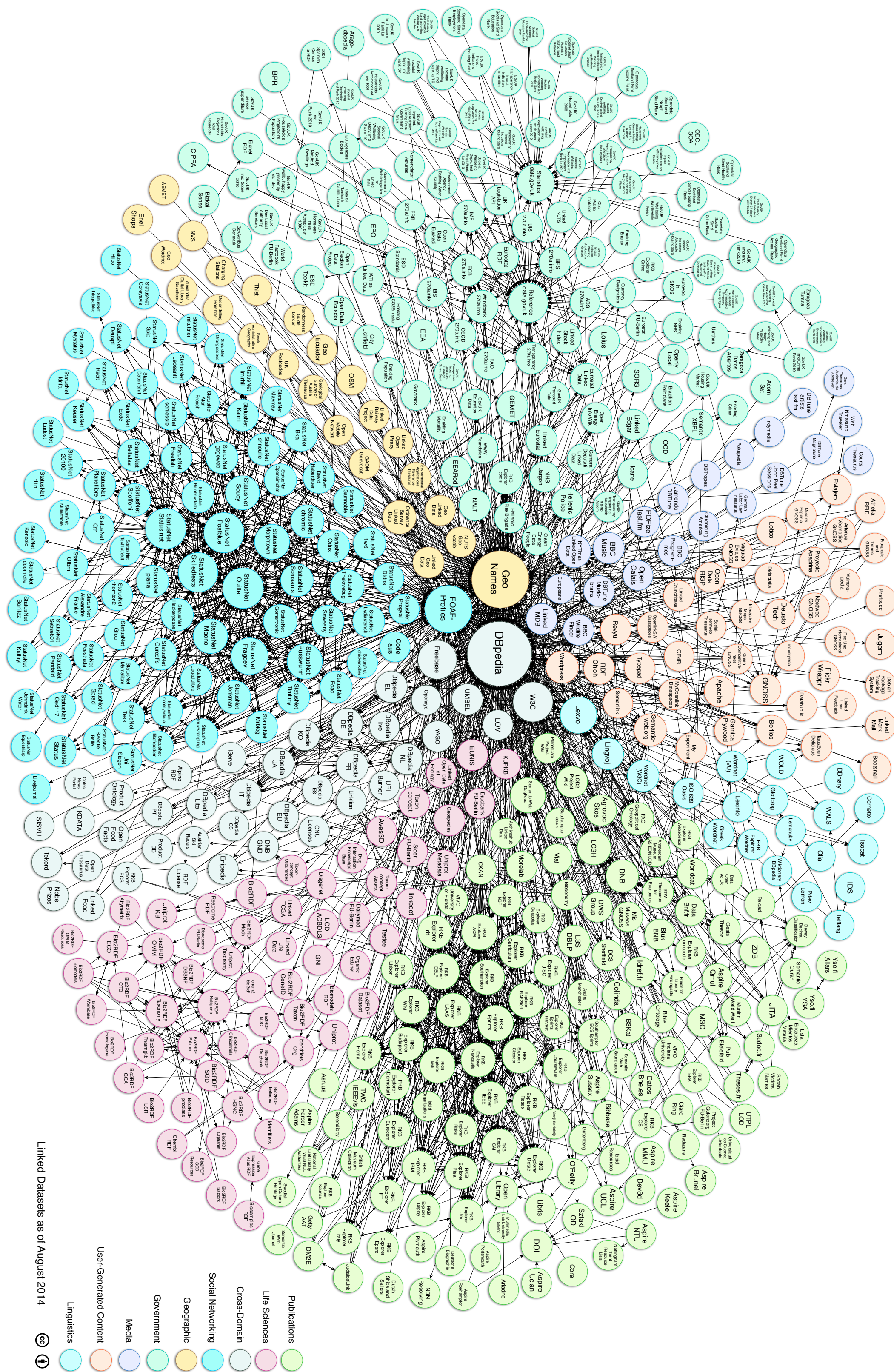


Abbildung 2: Linking Open Data Cloud Diagram 2014, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak. `http://lod-cloud.net` [SBP14]

3 WISSENSGENERIERUNG DURCH REASONING

Im einfachsten Fall kann sich eine semantische Wissensbasis wie eine klassische Datenbank verhalten, in der sich genau die Informationen abfragen lassen, die zuvor hinterlegt worden sind [BHL+14]. Um darüber hinaus die Möglichkeiten der formalen Beschreibung zu nutzen und auch auf implizit gegebene Fakten zuzugreifen, müssen diese materialisiert werden, was typischerweise Reasoning genannt wird. Reasoning bezeichnet somit im Allgemeinen das Ableiten oder Schließen von implizitem Wissen aus einer explizit gegebenen Faktenbasis [Fur14]. Eine einfache Faktenbasis kann z.B. die Informationen enthalten, dass Bob ein Professor ist und alle Professoren Personen sind. Ausgehend von diesen Informationen fällt es einem Menschen leicht, durch logisches Schließen abzuleiten, dass Bob eine Person sein muss. Die explizit gegebenen Fakten („Bob ist ein Professor" und „alle Professoren sind Personen") werden also durch die implizit enthaltenen Informationen ergänzt und können beispielsweise bei Anfragen an eine Wissensdatenbank so die gewünschten Ergebnisse vervollständigen.

Das folgende Kapitel gibt zunächst einen Überblick über die Grundlagen des Reasonings sowie eine kurze Einführung in verschiedene Reasoning Mechanismen. Im Anschluss folgt eine detaillierte Betrachtung von verwandten Arbeiten. Dabei wird bereits auf verschiedene Möglichkeiten der Lastverteilung und Parallelisierung für die Reasoner-Implementierung eingegangen.

3.1 REASONING GRUNDLAGEN

Die zu Beginn des Kapitels gegebenen Informationen, dass Bob ein Professor ist und alle Professoren auch Personen sind, reichen für eine maschinelle Schlussfolgerung zur Erweiterung der Wissensbasis um das Wissen, dass Bob eine Person ist, nicht aus. Stattdessen setzt das Schließen der implizit gegebenen Fakten durch eine Maschine bzw. einen Computer eine formale Wissensrepräsentation voraus, die auf Techniken der Logikprogrammierung [Llo87] basiert (vgl. Kapitel 2.1). In Abhängigkeit der notwendigen Expressivität einer Ontologie kommen vor allem Untermengen der Prädikatenlogik wie einfache Horn Logik [Hor51] oder Beschreibungslogiken [BN03] (im Folgenden auch „Description Logic" oder kurz DL) für die Beschreibung von Ontologien zum Einsatz. Die darauf aufbauenden Möglichkeiten des Reasonings sind vielseitig. Ein Anwendungsgebiet für Reasoner-Mechanismen liegt in der Entwicklungsphase neuer Ontologien, um stets die

Konsistenz des modellierten Wissens sicherzustellen [BHL+14]. Eine Inkonsistenz kann entstehen, wenn beispielsweise eine Klasse bzw. ein Konzept so umfangreich spezifiziert ist, dass niemals eine Instanz dieser Klasse existieren kann, da sich Eigenschaften der Klasse widersprechen. Als Ursache für die Entstehung inkonsistenter Ontologien werden in [KPSH05] drei wesentliche Gründe genannt. Im einfachsten Fall führt ein nicht ausreichendes Verständnis für die Anwendungsdomäne durch den Entwickler zu einer fehlerhaften Modellierung. Eine weitere Fehlerquelle kann während der Zusammenführung mehrerer Domain-Ontologien unter Verwendung des Import-Mechanismus (owl:import) entstehen, wenn die Entwickler der einzelnen Ontologien ein unterschiedliches Verständnis des modellierten Wissens haben oder konfliktbehaftete Konstrukte verwendet werden. Eine dritte Fehlerquelle liegt laut [KPSH05] in der fehlerhaften Überführung von anderen Datenquellen in OWL, die zu falschen Spezifikationen von Konzepten und Instanzen führen kann.

Eine weitere Möglichkeit, die Reasoner während der Entwicklung von Ontologien neben der Konsistenzprüfung bieten, ist die Abfrage nach Konzepten, die aufgrund ihrer Beschreibung exakt die gleichen Instanzen besitzen und sich somit lediglich in der Bezeichnung des Konzepts unterscheiden [Hor08]. Während dieser Umstand in einigen Fällen intendiert sein kann (z.B. wären die Konzepte *Bücherei* und *Bibliothek* äquivalent zu sehen), ist er in anderen Fällen ggf. auf grundlegende Fehler im Design der Wissensbasis zurückzuführen, weshalb eine Abfrage und Auswertung entsprechender Konzepte durch den Entwickler zu einer weiteren Qualitätssteigerung beitragen kann [Hor13]. Des Weiteren trägt die Verwendung von Reasonern zur korrekten (impliziten) Hierarchiebildung zwischen den modellierten Konzepten bei, was sowohl während des Entwicklungsprozesses als auch anschließend während der konkreten Nutzung einer Ontologie relevant ist.

Neben dem Einsatz von Reasonern in der Design-Phase von Ontologien kommen sie vor allem auch während der Nutzung semantischer Anwendungen zum Einsatz. Durch die Vervollständigung der Wissensbasis ermöglichen sie die Beantwortung komplexer Suchanfragen unter Berücksichtigung des implizit gegebenen Wissens. Auf diese Weise lassen sich Zusammenhänge sowohl innerhalb von Datensätzen als auch zwischen verschiedenen Datensätzen finden, die ohne das Reasoning verborgen bleiben. Somit stellt das Reasoning eine wesentliche Funktionalität für die Verwendung semantischer Technologien und bei der Entwicklung des semantischen Webs dar [KvHW11] [May06] [PBSZ14], durch die sich eine semantische Anwendung klar von klassischen Datenbanksystemen abhebt.

3.1.1 *Komplexität*

Bei dem Einsatz von Reasoning-Algorithmen spielen sowohl die Entscheidbarkeit als auch die Komplexität eine wichtige Rolle. Die Entscheidbarkeit beantwortet die Frage, ob es für eine beschriebene Semantik ein System geben kann, das in der Lage ist, nur korrekte und vollständige Schlüsse zu ziehen und das für jede Aussage feststellt, ob sie richtig oder falsch ist [PB06]. Die Komplexität beschreibt den notwendigen Aufwand unter Berücksichtigung von Zeit und Ressourcenverbrauch im Verhältnis zur Größe des Problems. Sie wird für konkrete Problemstellungen im Bereich des semantischen Webs durch die Verwendung von Komplexitätsklassen angegeben. Als akzeptabel wird dabei die polynomielle Komplexität PTIME (etwa n^c, mit n gleich der Größe der Eingabe und c einer Konstanten) gesehen, während eine exponentielle Komplexität NEXPTIME (etwa 2^n mit n gleich der Größe der Eingabe) häufig in Abhängigkeit der Problemgröße als in der Praxis nicht mehr nutzbar angesehen wird [W3C12]. Die Herausforderung bei der Entwicklung von Ontologiesprachen und dazugehörigen Reasoning-Algorithmen besteht also in der Erhaltung der Entscheidbarkeit und in der Beherrschung der Komplexität durch ein möglichst polynomielles Verhalten.

3.1.2 *Beschreibungslogik*

Description Logics [BN03] sind eine Familie von Sprachen zur Wissensrepräsentation, die zur Beschreibung von Zusammenhängen zwischen Konzepten einer Domäne [PB06] genutzt werden können. Die Basis hierfür wird durch atomare Konzepte wie z.B. Person, Dozent, Student, männlich oder weiblich gebildet. Eine grundlegende Eigenschaft von DLs liegt in der Beschreibung neuer, komplexer Konzepte auf der Basis bereits existierender. Beispielsweise kann das Konzept *Frau* unter Verwendung der Äquivalenz ($C \equiv D$) mittels des folgenden Ausdrucks beschrieben werden:

$$Frau \equiv Person \sqcap weiblich$$

Das Konzept *Frau* bezeichnet also alle Dinge, die sowohl *Person* als auch *weiblich* sind ($\sqcap$ steht für die Schnittmenge, während $\sqcup$ die Vereinigung bezeichnet). Neben den explizit definierten Konzepten existieren zusätzlich $\top$ für das häufig als *Thing* oder *universelle Konzept* bezeichnete Konzept sowie das leere Konzept $\bot$. Zusätzlich zur Definition von Konzepten können durch die Verwendung von Rollen die Eigenschaften von Konzepten beschrieben werden. Beispielsweise lässt sich unter Verwendung der existentiellen Quantifizierung mit dem Ausdruck

$$\exists betreut_student.männlich$$

das Konzept aller Dinge beschreiben, die mindestens einen männlichen Studenten betreuen. Der Allquantor ∀ hingegen erlaubt beispielsweise durch den Ausdruck

$$\forall betreut_student.männlich$$

die Beschreibung des Konzepts aller Dinge, die ausschließlich männliche Studenten betreuen.

Die zuvor beschriebenen Eigenschaften dienen der Beschreibung der TBox von DLs. Sie legen also die Terminologie der Wissensbasis fest, die durch die ABox ergänzt werden kann. Die ABox enthält konkretes Wissen über Instanzen, das beispielsweise wie folgt definiert werden kann:

$$\begin{aligned} Person(Bob) \\ weiblich \sqcap Person(Anna) \\ betreut_student(Bob, Anna) \end{aligned}$$

Konkret identifizieren diese Ausdrücke Bob im Allgemeinen als Person und Anna als eine weibliche Person. Zusätzlich wird die Information gegeben, dass Bob der Betreuer von Anna ist.

Die TBox einer DL dient nicht nur der reinen Schema-Definition wie in klassischen (relationalen) Datenbanksystemen, sondern auch dem Ableiten von Wissen (Reasoning). Bei einer Beschreibung des Konzepts *Vater* durch

$$(Vater \equiv \exists hat_Kind.\top)$$

in der TBox und $Vater(Bob)$ in der ABox würde beispielsweise die Antwortmenge auf die Frage nach Personen, die ein Kind haben, Bob beinhalten [PB06]. Die Komplexität des Reasoning-Prozesses ist dabei von der Ausdrucksstärke der TBox abhängig. Die Expressivität kann zum Beispiel durch die Einführung von Kardinalitäten für Rollen erhöht werden, die Ausdrücke wie

$$(\geqslant 3 betreut_student.\top)$$

erlauben, mit denen in diesem Fall alle Dozentinnen und Dozenten beschrieben werden, die mindestens drei Studenten betreuen. Eine zusätzliche Erweiterung stellen auch die *Rollenkonstruktoren* dar, die abgeleitete Rollen als Vereinigungs- oder Schnittmenge erlauben. Letztendlich charakterisieren sich die einzelnen Beschreibungslogiken über die Sprachelemente, die innerhalb der Logik zur Definition von komplexen Konzepten und Eigenschaften bereitgestellt werden [Pan04]. Zusätzliche Sprachelemente tragen zu einer größeren Expressivität bei, die jedoch auch in einer höheren Komplexität beim Reasoning resultiert. Auf diese Weise kann für Description Logics schnell eine NEXPTIME-Komplexität erreicht wer-

den, die hoch spezialisierte Algorithmen erfordern, um das Reasoning in einer annehmbaren Zeit zu erlauben [PB06].

3.2 REASONING-MECHANISMEN

Für den Zweck des semantischen Reasonings haben sich verschiedene Mechanismen etabliert. Grob lassen sich diese in DL-Reasoner und regelbasierte-Reasoner untergliedern, wobei sich insbesondere die DL-Reasoner z.B. entsprechend der verwendeten Algorithmik weiter kategorisieren lassen (beispielsweise Tableau Reasoning und Resolution basiertes Reasoning). Eine Kategorisierung in fünf Reasoner-Kategorien wird z.B. in [TKO13] vorgeschlagen, der eine Betrachtung von insgesamt 26 Reasonern voraus ging. Dabei lässt sich nicht immer eine klare Zuordnung treffen, da einige Ansätze auf hybriden Methoden aufsetzen.

Im Folgenden wird zunächst ein kurzer Einblick in das Tableau-Reasoning-Verfahren gegeben, das unter den DL-basierten Ansätzen am häufigsten genutzt wird [FU10]. Der Einblick dient der Verständnisbildung und erhebt keinen Anspruch auf Vollständigkeit oder auf eine tiefgreifende formale Einführung. Weitere Details betreffend des Tableau Reasonings können z.B. in [BS01] nachgelesen werden. Im Anschluss folgt eine Einführung in das regelbasierte Reasoning. Dieser Ansatz wird im weiteren Verlauf der Arbeit vertieft und dient als Grundlage für weitere Entwicklungen. Der Grund für die Fokussierung auf regelbasiertes Reasoning liegt zum einen in der Zielsetzung dieser Arbeit begründet, die explizit eine Definition der anzuwendenden Semantik in Form von Regeln fordert. Zum anderen adressiert die vorliegende Arbeit vor allem Large-scale-Datensätze, die häufig auch eine große ABox, also viele Instanzdaten, aufweisen. Für diese Bedingungen eignen sich besonders regelbasierte Reasoner [BHJV08], während sich DL-Reasoner häufig schwer skalieren lassen und ein effizientes Reasoning über große Instanzdaten nur mit Semantik-spezifischen Optimierungen gelingt [FU10] [FvH07] [HLTB04] [MS06].

3.2.1 *Tableau Reasoning*

Eine Möglichkeit der Implementierung von Reasoning-Mechanismen basiert direkt auf der Description Logic und wurde im vorherigen Kapitel bereits angedeutet. In der Logik-Programmierung und insbesondere auch bei der Verwendung von Description Logic bedient man sich häufig dem Tableau-Beweisverfahren zum Ableiten von implizit gegebenen Wissen.

Der Tableau-Algorithmus basiert im Wesentlichen auf dem Finden von Widersprüchen in einer Wissensbasis. Ausgehend von einer Frage, ob aus einer Aussage C eine Aussage D folgt, geht der Algorithmus zunächst davon aus, dass die an-

gestrebte Behauptung falsch ist, es also einen Fall gibt, in dem C erfüllt ist und D nicht. Dazu werden aus einer gegebenen Wissensbasis so lange logische Konsequenzen erzeugt, bis ein Widerspruch gefunden wird oder keine weiteren Konsequenzen mehr abgeleitet werden können. Ein Widerspruch entsteht z.B. dann, wenn sowohl $C(a)$ als auch $\neg C(a)$ in einer Wissensbasis vorliegen. Ein einfaches Beispiel für den Ablauf des Beweisverfahrens ist in [HKRS08] gegeben und geht von der Wissensbasis W mit den folgenden Aussagen aus:

$$C(a)$$
$$(\neg C \sqcap D)(a)$$

Außer Frage steht die logische Konsequenz $C(a)$ aus W. Zusätzlich kann die Aussage $(\neg C \sqcap D)(a)$ in die prädikatenlogische Formel $\neg C(a) \wedge D(a)$ überführt werden, woraus wiederum $\neg C(a)$ abgeleitet werden kann [HKRS08]. Entsprechend ist in der Wissenbasis sowohl $C(a)$ als auch $\neg C(a)$ enthalten, was zu einem Widerspruch führt und die Wissenbasis ungültig macht. Das Vorgehen bei komplexeren Formeln ist ähnlich. Jede Formel wird immer weiter in ihre einzelnen Bestandteile zerlegt, wodurch ein Baum entsteht, bei dem jeder Ast eine Widerspruchs-Prüfung darstellt.

Während die worst-case-Komplexität eines naiven Tableau-Algorithmus sehr hoch ist, kann in der Praxis durch den Einsatz von optimierten Verfahren eine gute Leistung erzielt werden [SS09], die für viele Anwendungsfälle ausreichend ist. Das betrifft vor allem Szenarien, in der eine große TBox und eine verhältnismäßig kleine ABox zum Einsatz kommen. Weniger geeignet ist der Reasoning-Mechanismus hingegen bei der Verarbeitung einer großen Menge von Instanz-Daten [FU10], wie sie vor allem im Zusammenhang mit dem Semantic Web auftreten. Ein weiterer Nachteil des Tableau-Algorithmus besteht darin, dass sich die Schlussregeln ausschließlich aus der Semantik der verwendeten Beschreibungslogik ergeben (aus den Konzept- und Rollendefinitionen) und keine anwendungsabhängigen Ableitungsregeln möglich sind [PB06].

3.2.2 *Regelbasiertes Reasoning*

Das vorherige Kapitel hat einen Überblick über Description Logic Reasoning, basierend auf dem Tableau-Algorithmus, gegeben. Eine weitere Möglichkeit für das Ableiten implizit gegebener Fakten aus einer Wissensbasis besteht in der Verwendung von Regel-Systemen bzw. Horn-Logik [HJS11], die sich ebenfalls aus einer Teilmenge der Prädikatenlogik ableiten lässt [AH08]. Somit stellen Regelsysteme eine weitere Möglichkeit der Wissensrepräsentation dar.

Eine Regel hat die Form

$$B \leftarrow A_1 \wedge ... \wedge A_n$$

wobei A_i und B atomare Ausdrücke sind. Die Interpretation der Regel lässt zwei Varianten zu, die als *deduktiv* und *reaktiv* beschrieben werden. Dem deduktiven Schließen liegt die Annahme zugrunde, dass wenn $A_1, ..., A_n$ als wahr bekannt sind, auch B als wahr angenommen werden kann. Die reaktive Interpretation hingegen überprüft die Bedingungen $A_1, ..., A_n$ und führt im Falle einer positiven Auswertung (alle Bedingungen sind wahr) die Aktion B aus. Im Sinne der Wissensgenerierung wird die deduktive Interpretation angewandt, die zur Erweiterung der Wissensbasis führt. Die Regel

$$Person(y) \leftarrow Person(x) \wedge vater(x, y)$$

kann beispielsweise so gelesen werden, dass für jedes x und jedes y (x und y werden dabei jeweils als Variablen verstanden) die Aussage getroffen werden kann, dass, wenn x eine Person ist und x der Vater von y ist, dann ist auch y eine Person. Somit kann die Wissensbasis um $Person(y)$ ergänzt werden [PSH07]. Gleichzeitig zeigt das Beispiel, dass sich Regeln dazu eignen, zusätzliches Wissen zu formalisieren und dazu genutzt werden können, eine Faktenbasis zu vervollständigen bzw. zu ergänzen.

Die Anwendung von Regeln auf einen Datensatz kann, wie zu Beginn dieser Arbeit bereits eingeführt, zu verschiedenen Zeitpunkten stattfinden. Eine insbesondere bei großen Datenmengen häufig verwendete Methode ist das *Forward-Chaining* (auch *Bottom-Up* genannt). Bei dieser Variante wird vor der Weiterverarbeitung, z.B. der Beantwortung von Queries, die Semantik auf den gesamten Datensatz angewandt, um sämtliche mögliche Aussagen zu materialisieren [Abr05] [FU10]. Der daran anknüpfenden Verarbeitung liegt so ein vollständig materialisierter Datensatz vor, was beispielsweise eine schnelle Beantwortung von Queries erlaubt. Als nachteilig anzusehen ist jedoch der initiale Aufwand zur Verarbeitung des gesamten Datensatzes, unabhängig von dem tatsächlich benötigten Teilausschnitt der Faktenbasis. In diesem Punkt unterscheidet sich das Forward-Chaining vom *Backward-Chaining*, das nur die benötigten Teilausschnitte eines Datensatzes zur Anfragezeit materialisiert. Dazu werden ausgehend von einer Anfrage Regeln und Aussagen identifiziert, die zur Beantwortung der Anfrage beitragen können. Dieser auch als *Top-Down* bezeichnete Ansatz geht also von dem Zielobjekt aus und leitet nur die für die Anfrage benötigten Informationen ab [Abr05]. Trotz der Einschränkung der ausschließlichen Berechnung der benötigten Daten ist die Vorgehensweise des Backward-Chaining häufig für interaktive Anwendungen in Bezug auf die Antwortzeiten nicht effizient genug, weshalb der Ansatz in diesen Fällen

auf kleine Datensätze bzw. eine einfach zu berechnende Semantik beschränkt ist [Urb13].

3.3 VERWANDTE ARBEITEN

Neben den klassischen DL-Reasonern haben sich verschiedene andere Ansätze für das Reasoning etabliert, die z.B. auf Datenbanksysteme aufsetzen [KWE10] [WED+08] oder Regelsysteme nutzen [JS04] [KR03] [BCC+10] [BCMS10]. Während zunächst oft die Implementierung der reinen Semantik und die Ausführung auf einzelnen Prozessoren im Vordergrund stand, haben sich mit dem Aufkommen des semantischen Webs vor allem Ansätze entwickelt, die auf einer parallelen Ausführung aufbauen. Entsprechend der Zielsetzung der vorliegenden Arbeit werden im Folgenden Ansätze des regelbasierten Reasonings mittels Production Systems sowie des parallelen Reasonings betrachtet. Als *Production System* [New73] werden im allgemeinen Systeme verstanden, die eine Menge von Regeln (Productions) auf eine Datenbasis anwenden, um z.B. weitere Aktionen auszuführen oder neue Daten abzuleiten. Somit stellen Production Systems die grundlegende Idee der regelbasierten Verarbeitung dar, wie sie auch in dieser Arbeit für die Definition der anzuwendenden Semantik verstanden wird.

Die Betrachtung der verwandten Arbeiten wird am Ende des Kapitels noch einmal zusammengefasst und für eine Abgrenzung zur eigenen Arbeit genutzt. Des Weiteren wird im Verlauf der Arbeit, insbesondere bei der in Kaptitel 10 beschriebenen Evaluation, auf die hier aufgeführten Ansätze verwiesen.

3.3.1 *Regelbasiertes Reasoning*

Das regelbasierte Reasoning, wie es im Gegensatz zur Implementierung eines bestimmten Regelsatzes in dieser Arbeit verstanden wird, zeichnet sich durch die Definition der anzuwendenden Semantik während der Ausführungszeit durch Regeln aus. Entsprechend muss die Implementierung des Reasoners unabhängig von einer bestimmten Semantik sein und z.B. die Möglichkeit bieten, benutzer- oder anwendungsabhängige Regeln anzuwenden. Klassische Expertensysteme, die auch zur Implementierung von Fachlogik genutzt werden, unterstützen diese Eigenschaft und stellen die Grundlage für verschiedene Arbeiten dar, die sich auf die Implementierung regelbasierter, semantischer Reasoner konzentrieren.

Erste Arbeiten wurden z.B. in [KR03] und [JS04] veröffentlicht, die auf der Java Expert System Shell (Jess) [FH+08] sowie auf Bossam, einem Expertensystem mit erweiterten Möglichkeiten für das Reasoning, basieren. Die verwendeten Expertensysteme implementieren jeweils den RETE-Algorithmus, der in Kapitel 4 im Detail vorgestellt wird. Um das Reasoning auf RDF Daten zu erlauben, fin-

det in beiden Systemen eine Transformation von Tripeln zu Fakten statt, die von dem jeweiligen Regelsystem verarbeitet werden. Bossam implementiert zusätzlich weitere semantikspezifische Funktionen, um eine möglichst große Abdeckung der OWL-Semantik zu erzielen. Bei der Evaluation des Systems, das basierend auf Queries Regeln zur Ausführung ableitet, steht die Vollständigkeit der Ergebnisse im Vordergrund, während die Ausführungsgeschwindigkeit nicht betrachtet wird. Auch die Ergebnisse in [KR03] geben keinen Aufschluss über die Größe der Daten, die verarbeitet werden können, noch lassen sich Rückschlüsse auf die Performance ziehen.

In [MB08] wird O-DEVICE vorgestellt, ein auf dem CLIPS Expertensystem aufsetzender Reasoner, der entsprechend der zugrunde liegenden Technologien auf einem objektorientierten Design basiert. Für das Reasoning in der OWL 1 Lite Semantik werden Ontologien in ein objektorientiertes Modell überführt, das gleichzeitig dazu genutzt wird, für eine effizientere Ausführung die anzuwendenden Regeln in Abhängigkeit der Ontologieeigenschaften abzuleiten. Die Evaluation über Daten mit bis zu 265k Tripeln zeigt eine im Gegensatz zu OWLIM [KOM05] langsamere Ausführung, für die jedoch weniger Speicher benötigt wird. OWLIM [KOM05] bietet sowohl eine Speicher- als auch eine Reasoning-Infrastruktur, die durch Regeln konfiguriert werden kann. Durch die vollständige Verarbeitung der Daten im Hauptspeicher kann für die in der Evaluation [KOM05] gezeigte Verarbeitung der bis zu 10M Tripel großen Datensätze eine für die Zeit der Veröffentlichung hohe Performance erreicht werden.

Eine aktuellere Arbeit, die das Reasoning mittels Expertensystemen beschreibt, wird in [BCC+10] und [BCMS10] vorgestellt. Bei dem Ansatz handelt es sich um eine hybride Architektur, die einerseits Pellet [PS04] als DL-Reasoner über OWL Ontologien nutzt, und andererseits das Business Rule Management System Drools[1], das durch Erweiterungen das Gesamtsystem zur Anwendung von Fuzzy-Logik befähigt.

Weiterhin wird die Technologie des regelbasierten Reasonings zur Implementierung spezieller Reasoner für ressourcenbeschränkte Geräte wie Smartphones und Sensoren innerhalb von Sensornetzwerken genutzt [SS11b] [TBKO09] [TKO11]. In [SS11b] wird der Einsatz der Production Rule-Engine CLIPS, die ebenfalls bei dem bereits erwähnten Reasoner O-Device [MB08] zur Anwendung kommt, auf einem embedded Hardware Gerät mit 64MB Speicher gezeigt. Die Wahl für einen regelbasierten Ansatz wird mit der hohen Komplexität und dem damit verbundenen Ressourcenverbrauch von DL-Reasonern begründet. Eine über eine reine Nutzung eines regelbasierten Reasoners hinausgehende Arbeit wird in [TKO11] vorgestellt. Um die Ausführung des genutzten RETE-Algorithmus zu optimieren, werden in [TKO11] im Wesentlichen zwei Maßnahmen durchgeführt. Zunächst

1 http://www.drools.org/

findet eine Auswahl an anzuwendenden Regeln aus dem gegebenen Regelsatz statt, die auf den in der zu verarbeitenden Ontologie genutzten OWL-Konstrukten basiert. Durch die gezielte Auswahl können in vielen Fällen die Anzahl der anzuwendenden Regeln und somit auch die Komplexität und der Speicherverbrauch des RETE-Algorithmus reduziert werden. Des Weiteren wird ein Zwei-Phasen RETE-Algorithmus eingeführt, der nach einer ersten Teil-Ausführung des Algorithmus (Phase 1) das zugrunde liegende RETE-Netzwerk für die häufig rechenintensivere Phase 2 optimiert, um die Anzahl der durchzuführenden Berechnungen möglichst gering zu halten. Eine detaillierte Betrachtung der beschriebenen RETE-Optimierung erfolgt in Kapitel 4 nach der Einführung des Algorithmus.

3.3.2 *Paralleles Reasoning*

Im Hinblick auf das semantische Web und der wachsenden Größe der verfügbaren RDF Datensätze lässt sich die Menge der semantischen Informationen nicht mehr mit klassischen Reasoner-Ansätzen bewältigen [LZK+09]. Eine Möglichkeit, der Herausforderung, die mit der Verarbeitung großer Datenmengen einher geht, zu begegnen, besteht in der Parallelisierung des Reasoning-Prozesses. Die Parallelisierung kann sich dabei sowohl auf die Nutzung der parallelen Strukturen moderner Computersysteme stützen (z.B. Mehrkernprozessoren), als auch auf die Verteilung der Rechenlast auf verschiedene Rechnerknoten. Beide Vorgehensweisen sind mit verschiedenen Vor- und Nachteilen verbunden. Während die Verteilung von Rechenlast auf verschiedene Knoten theoretisch beliebig horizontal skaliert werden kann (es können beliebig viele Rechenknoten und damit Rechenleistung hinzugefügt werden), erhöht sich mit jedem Knoten gleichzeitig auch der Koordinations- und Synchronisationsaufwand. Die Nutzung paralleler Hardware auf einzelnen Computern hingegen ist beschränkt in Bezug auf die verfügbaren Ressourcen und lässt sich somit nicht beliebig skalieren, verspricht aber aufgrund einer einfacheren Synchronisation von Prozessen einen geringeren Overhead und damit eine schnellere Ausführung.

Im Folgenden werden zunächst verwandte Arbeiten zu beiden Varianten der Parallelisierung getrennt voneinander betrachtet.

3.3.2.1 *Verteiltes Reasoning*

Erste Arbeiten zur Lastverteilung für semantisches Reasoning unter Verwendung von regelbasierten Ansätzen wurden 2008 von Soma und Prasanna vorgestellt [SP08b] [SP08a]. In ihrer Arbeit beschreiben sie verschiedene Möglichkeiten der Daten- und Regelpartitionierung, auf deren Basis eine parallele Verarbeitung auf mehreren Rechnern ermöglicht wird. Die vorgestellten Methoden der Partitionierung werden unter dem Aspekt der Minimierung von mehrfach ausgeführ-

ten Berechnungen sowie der Reduzierung des Kommunikationsaufwands zwischen verschiedenen Prozessen für die pD* Semantik betrachtet. Basierend auf der Datenpartitionierung und des verteilten Reasonings auf 16 Rechenknoten erreichen sie eine bis zu 18-fache Ausführungsgeschwindigkeit für Datensätze mit bis zu einer Millionen Tripel. Ein anderer Ansatz des verteilten Reasonings wird in [KMK08] beschrieben, der auf verteilte Hash-Tabellen (Distributed Hash Tables, DHTs) aufsetzt. Es werden sowohl Forward-Chaining- als auch Backward-Chaining-Reasoning unter Verwendung von DHTs betrachtet. Die Ergebnisse zeigen vor allem für das Forward-Chaining eine schlechte Skalierung, weshalb eine Verarbeitung von maximal 10k Tripel gezeigt wird.

In [OKA+09a] [OKA+09b] stellen die Autoren MARVIN vor, eine Plattform zur verteilten und parallelen Verarbeitung von großen RDF Datensätzen. Der vorgestellte Ansatz basiert auf einer „Divide-Conquer-Swap" Strategie, einer erweiterten Divide-and-Conquer Strategie. Dabei wird ein Datensatz von Rechnern, die über eine Peer-to-Peer-Infrastruktur verbunden sind, in verschiedene Partitionen geteilt, die einzeln verarbeitet werden, um so neue Tripel abzuleiten. Im Anschluss wird eine Neupartitionierung vorgenommen und die Ergebnisse an andere Rechenknoten übergeben. Dieser Vorgang wird wiederholt, bis keine neuen Tripel mehr erzeugt werden. Die Evaluation erfolgt auf einem Cluster mit insgesamt 271 Knoten und Datensätzen mit einer maximalen Größe von 14.9 Millionen Tripeln. Aufbauend auf den Ergebnissen aus [OKA+09b] [OKA+09a] wird in [KOvH10] der Ansatz der Datenverteilung durch die Einführung selbstorganisierender *elastischer Regionen* erweitert. Statt einer deterministischen Verteilung von Tripel auf die verschiedenen Rechenknoten findet die Verteilung in Abhängigkeit der vorliegenden Daten satt. Tripel, die Gemeinsamkeiten in Form eines Terms besitzen, werden innerhalb einer Region gesammelt, um den Reasoning-Prozess jeweils auf einem zusammenhängenden Teilausschnitt des Datensatzes durchzuführen. Die Evaluation zeigt das auf bis zu 64 Rechnern verteilte RDFS Reasoning von Datensätzen mit bis zu 200 Millionen Tripel, wobei ein Durchsatz von bis zu 450k Tripel pro Sekunde (ktps) erreicht wird.

Der Ansatz der Datenpartitionierung zur parallelen Verarbeitung wird auch in [PGSH14] gewählt. Anders als bei der MARVIN-Plattform wird ein Datensatz jedoch in Partitionen unterteilt, die ein vollständig voneinander unabhängiges Reasoning für die OWL-Lite Semantik erlauben. Die erstellten Partitionen werden anschließend auf verschiedenen Rechnern parallel verarbeitet, ohne dass eine Kommunikation zwischen den Rechnern notwendig ist.

Neben der Verwendung von Peer-to-Peer-Netzwerken [OKA+09b] und verteilten Hash Tabellen [KMK08] zur Implementierung skalierbarer Reasoner-Architekturen wird in vielen Arbeiten ein MapReduce [DG04] basierter Ansatz gewählt. MapReduce ist ein Programmiermodell, das sich besonders zur parallelen Ver-

arbeitung großer Datenmengen auf einem Rechencluster eignet. Es basiert auf der Ausführung aufeinander folgender *Map* und *Reduce* Methoden, die vom Entwickler programmiert werden müssen. Während der Map Phase wird aus den Eingabedaten ein Set aus Key-Value-Paaren erzeugt, die anschließend durch die Ausführung der Reduce Methode weiter zusammengefasst werden. Beide Methoden können dabei jeweils parallel auf mehreren Rechnern gleichzeitig ausgeführt werden. Die zugrunde liegenden Operationen zur Datenverteilung, Parallelisierung, Kommunikation zwischen den Rechnerknoten und der Lastverteilung werden vollständig vom MapReduce-Framework übernommen, was die Entwicklung skalierbarer, verteilter Anwendungen stark vereinfacht [DG04].

Erste Arbeiten unter Verwendung von MapReduce wurden von Urbani et al. [UKOH09] sowie von Weaver et al. [WH09] veröffentlicht. In [UKOH09] stellen die Autoren verschiedene Optimierungen für die Implementierung der RDFS Semantik durch das MapReduce Programmiermodell vor, durch die eine skalierbare Ausführung erreicht wird. Die Optimierungen basieren auf unterschiedlichen Ansätzen, wie dem Vorhalten von Schema-Tripeln im Speicher, der Gruppierung der Eingabedaten sowie der Sortierung der anzuwendenden Regeln. Des Weiteren wird die RDFS Semantik auf das Subset ρdf [MPG07] reduziert, das alle Regeln mit nur einem Regel-Term ausschließt, da sich diese zu jeder Zeit unabhängig von anderen Tripeln berechnen lassen [UKOH09]. Die Evaluation erfolgt über Datensätze mit einer Größe von insgesamt 865M Tripeln auf 64 Rechnern.

Im Gegensatz zu [UKOH09] beschreiben Weaver et al. in [WH09] die vollständige Anwendung der RDFS Semantik, die durch eine Ausführung auf 128 Rechnern mit Datensätzen von bis zu 346M Tripeln evaluiert wird. Dazu werden die Schema-Tripel auf jeden Rechenknoten dupliziert, während die Instanz-Tripel durch die Aufteilung in verschiedene Partitionen auf unterschiedliche Rechner verteilt werden. Der eigentliche Reasoning-Prozess findet auf den einzelnen Rechnern durch klassische Reasoner-Implementierungen statt.

Aufbauend auf der Arbeit aus [UKOH09] führen Urbani et al. in [UKM+10] und [UKM+12] weitere Semantik-spezifische Konzepte ein, um nicht nur RDFS Reasoning, sondern auch pD* Reasoning effizient durch das MapReduce Programmiermodell ausführen zu können. Das vorgestellte System WebPIE (Web-scale Parallel Inference Engine) wird mit Datensätzen mit einer Größe von bis zu 100 Milliarden Tripeln evaluiert. Die Evaluation zeigt eine annähernd lineare Skalierung bei der Verwendung von 64 Rechnern im Cluster und einer steigenden Datensatzgröße (bei gleichbleibender Komplexität) [UKM+12].

Weitere Arbeiten, die das Programmiermodell MapReduce nutzen, beschäftigen sich zum Beispiel mit der Implementierung anderer, teils DL-basierter Semantik [MMH10] [SS11a] sowie mit der Einbeziehung von Fuzzy Reasoning [LQWY11] [LQWY12] [ZQL+12].

Die Zielsetzung der in [WWAH10] vorgestellten Arbeit adressiert nicht wie die zuvor genannten MapReduce basierten Ansätze ausschließlich die Implementierung eines skalierbaren Reasoners, sondern auch die Möglichkeit der anschließenden Verarbeitung der Daten auf klassischen Rechnern. Anders als in der vorliegenden Arbeit, in der ein ganzheitliches System für semantisches Reasoning großer Datenmengen auf einem einzelnen Rechner entwickelt werden soll, wird in [WWAH10] eine Kombination aus verschiedenen Ansätzen genutzt. Dazu werden Methoden des parallelen Reasoning sowie der parallelen Query-Verarbeitung auf mehreren Rechnern eingesetzt, um für einen Benutzer bzw. eine Benutzerin relevante Teilausschnitte eines Datensatzes zu bilden. Während im ersten Schritt durch einen verteilten Reasoner-Prozess ein Datensatz entsprechend einer Semantik vollständig materialisiert wird, wird im zweiten Schritt basierend auf einer SPARQL-Abfrage ein Teilausschnitt gebildet. Die Bildung von Teilausschnitten dient der Reduzierung der verfügbaren Informationen für eine einfachere Weiterverarbeitung. Im letzten Schritt wird der extrahierte Teilausschnitt, basierend auf dem BitMat-Verfahren [ACZH10] [AH09], komprimiert, um die Speicherbelastung (sowohl in Bezug auf den Hauptspeicher, als auch auf die Festplatte) während der weiterführenden Bearbeitung durch den Benutzer möglichst gering zu halten. Das BitMat Verfahren erlaubt eine Repräsentation von Tripeln in Form von binären (komprimierten) Matrizen, die sich auch für eine anschließende Query-Auswertung eignen. Der so komprimierte Teilausschnitt kann anschließend auf Standard-Hardware genutzt werden, um weitere Abfragen (beschränkt auf den Teilausschnitt) auszuführen. Die in [WWAH10] gezeigte Evaluation wird auf rund 900M Tripeln ausgeführt und nutzt für den Reasoning-Prozess bis zu 1788 parallele Prozesse innerhalb eines Rechner-Clusters, die die RDFS Semantik in rund 30 Minuten anwenden. Die Reduktion auf einen relevanten Teilausschnitt wird auf einem Supercomputer mit 8192 Rechenknoten ausgeführt und nimmt rund 16 Minuten in Anspruch.

3.3.2.2 *Reasoning unter Verwendung (massiv) paralleler Hardware*

Während im vorherigen Kapitel Reasoner-Ansätze betrachtet wurden, die eine Parallelität durch die Verteilung der Rechenlast auf mehrere Rechenknoten innerhalb eines Clusters erzielten, steht in diesem Kapitel die Nutzung von parallelen Strukturen einzelner Rechner im Fokus. In [GM10] und [GJM+11] wird ein In-Memory Ansatz für das Reasoning mit der ρdf Semantik auf einem Cray XMT Supercomputer vorgestellt. Die in [GM10] verwendete Cray XMT Konfiguration unterstützt den gemeinsamen Zugriff von 512 *Threadstorm* Prozessoren auf insgesamt 4 TB Speicher. Ein Threadstorm Prozessor unterstützt jeweils 128 parallele Threads, wodurch sich die Architektur mit der Gesamtzahl der zeitgleich ausführbaren Threads insbesondere für eine feingranulare parallele Ausführung eignet.

Der große Hauptspeicher ermöglicht eine auf verteilten Hash-Tabellen basierende Organisation und Speicherung aller Tripel im globalen Speicher, die neben schnellen Zugriffszeiten auch eine Möglichkeit der Erkennung von mehrfach abgeleiteten Tripeln (Duplikaten) bietet. Der Reasoning-Prozess basiert auf einer Semantikspezifischen Ausführung sowie entsprechenden Optimierungen. Dazu wurde in [GM10] für jede Regel eine Hash-Queue erstellt, um Zwischenergebnisse für die jeweiligen Regeln (dem Regelmuster entsprechende Tripel) abzulegen. In [GJM+11] wurden diese Strukturen zugunsten des Speicherverbrauchs aufgehoben und ein Anstieg der Rechenzeit bei einer Nutzung von 128 Prozessoren um 11% akzeptiert. Insgesamt konnte das Reasoning auf bis zu 20 Milliarden Tripeln unter ausschließlicher Nutzung des Hauptspeichers gezeigt werden, wobei ein Durchsatz von bis zu 13.7 Millionen Tripel pro Sekunde erzielt wurde.

Neben der parallelen Hardware, die z.B. Supercomputer bieten, weisen vor allem moderne Central Processing Units (CPUs) und Graphics Processing Units (GPUs) eine immer leistungsstärkere und zum Teil hoch parallele Architektur auf. Diese wird z.B. in [HP12] genutzt, um einen Reasoner für die ρdf Semantik zu entwickeln. Ebenso wie bei [GJM+11] werden sämtliche Berechnungen auf Dictionary-kodierten Tripeln ausgeführt (es werden numerische Repräsentationen für die einzelnen Tripel-Elemente genutzt, siehe Kapitel 9.1) und semantikspezifische Optimierungen vorgenommen. Die Evaluation erfolgt auf einem Server mit vier CPUs, die jeweils acht Kerne aufweisen und auf einem Rechner, der mit einer AMD Radeon Grafikkarte ausgestattet ist. Gegenüber der Ausführung auf den CPUs kann durch die Verwendung der GPU für die bis zu 36 Millionen Tripel großen Datensätze kein Geschwindigkeitsvorteil erzielt werden, was auf die hohen Kosten für den Transfer von Speicherobjekten zwischen dem Hauptspeicher bzw. dem Prozessor und der Grafikkarte zurückzuführen ist [HP12]. Des Weiteren ist der in [HP12] vorgestellte Ansatz beschränkt auf die ρdf Semantik sowie auf Datensätze, die vollständig in den Speicher der verwendeten Grafikkarte geladen werden können (in numerischer Repräsentation).

Weitere Arbeiten, die auf der Nutzung von parallelen Architekturen moderner Computer aufbauen, sind z.B. verschiedene Description Logic Reasoner [KKS11] [MNP+14b] sowie Ansätze des Stream-Reasonings [UMJ+13] [LUQ14]. Der ELK Reasoner beispielsweise [KKS11] [KKS12] [KKS14] ist ein DL-Reasoner für das OWL-EL Profil und adressiert damit eine spezielle Ontologiesprache. Ein Performancevorteil gegenüber anderen Reasonern mit einer ähnlichen Sprachunterstützung wird vor allem durch die Nutzung mehrerer Threads erzielt, die bei einer Verarbeitung auf einem Mehrprozessor- oder Mehrkernsystem parallel ausgeführt werden können. Die Parallelisierung wird entsprechend durch sprachabhängige Optimierungen erreicht. Ein ähnliches Konzept wird bei dem RDFox Reasoner [MNP+14a] [MNP+14b] angewandt. Im Gegensatz zu ELK unterstützt er

das OWL-RL Profil und basiert weitestgehend auf Lock-freien Datenstrukturen, die bei einer parallelen Datenverarbeitung durch mehrere Threads das Abrufen der verfügbaren Rechenleistung von Mehrkernprozessoren erlaubt. Die Parallelität während der Ausführung wird erreicht, indem eine feste Anzahl an Threads erzeugt wird, die für die Verarbeitung der Tripel zuständig sind. Sofern noch vorhanden, ruft jeder Thread ein noch nicht verarbeitetes Tripel zur Iteration durch die verschiedenen Regeln ab. Kann dabei ein neues Tripel erzeugt werden, wird dieses direkt in den Datenbestand aufgenommen. Die Evaluation erfolgt mittels verschiedener Regelsätze auf bis zu 691M Tripeln. Durch die Verwendung von 32 Threads kann eine bis zu 19.5-fache Ausführungsgeschwindigkeit gegenüber der Ausführung durch lediglich einen Thread erzielt werden.

Im Gegensatz zu den bisher betrachteten Ansätzen wird beim Stream-Reasoning ein Eingabestream bestehend aus RDF Daten verarbeitet, was das Reasoning über dynamische Daten erlaubt. Während die in dieser Arbeit angestrebte Vorgehensweise eine Verarbeitung statischer Daten vorsieht, können die Arbeiten dennoch Gemeinsamkeiten in Bezug auf die Materialisierung neuer Tripel aufweisen. In [UMJ+13] stellen Urbani et al. DynamiTE vor, ein Stream-Reasoner, der die parallele Materialisierung über Eingabestreams adressiert. In der vorgestellten Arbeit wird die ρdf Semantik basierend auf Regeln implementiert. Aufbauend auf einer Einteilung in drei Kategorien, die entsprechend der Eigenschaften einer Regel vorgenommen wird, werden die anzuwendenden Regeln parallel durch mehrere Threads bearbeitet. Die Kategorisierung legt gleichzeitig die Ausführungsreihenfolge fest, so dass Regeln unterschiedlicher Kategorien nacheinander ausgeführt werden. Entsprechend der beschriebenen Architektur wird die Parallelität moderner CPUs genutzt, wobei der Grad der Parallelisierung bzw. die Anzahl der parallel ausgeführten Threads von der Regelbasis abhängig ist. Bei der Ausführung des Reasoning-Prozesses auf statischen Daten wurde für DynamiTE ein Durchsatz von 227 ktps (bezogen auf die Eingabedaten) erreicht [UMJ+13]. Weitere Ideen für das Stream-Reasoning unter Verwendung von GPUs wurden bereits in [LUQ14] skizziert. Eine Vertiefung der Arbeit durch konkrete Algorithmen und eine Evaluation des Ansatzes stehen noch aus.

3.3.3 *Limitierung bestehender Reasoner-Ansätze*

Die relevanten Arbeiten im Kontext des regelbasierten Reasonings legen im Wesentlichen zwei Schwerpunkte. Einerseits werden existierende Expertensysteme für den Reasoning-Prozess genutzt, um eine möglichst hohe Effizienz auch für große Datenmengen zu erreichen und das Reasoning ggf. um Möglichkeiten des Umgangs mit Unsicherheiten (Fuzzy) zu ergänzen. Die in diesem Kontext genannten Arbeiten unterscheiden sich im Wesentlichen durch die für die jeweilige Eva-

luation verwendeten Regeln sowie die zugrunde liegenden Expertensysteme. Eine tatsächliche Verarbeitung großer Datenmengen, mit mehreren hundert Millionen Tripeln und mehr, wie sie in dieser Arbeit unter *Large-scale* verstanden werden, konnte jedoch nicht gezeigt werden. Ein zweiter Schwerpunkt der Verwendung von regelbasierten Reasonern liegt in der Anwendung auf ressourcenbeschränkten Geräten. Hier konnten insbesondere in [TKO11] durch die Verwendung eines optimierten RETE-Algorithmus Ergebnisse erzielt werden, die sich von der Ausführungsgeschwindigkeit einer Standardimplementierung abheben.

Eine Beschränkung der regelbasierten Ansätze besteht jedoch in der ausschließlichen Verwendung einer seriellen Ausführung. Keiner der Ansätze verfolgt eine Parallelisierung, um die Effizienz zu steigern und z.B. große Datenmengen besser bewältigen zu können oder einen Geschwindigkeitsvorteil zu erzielen.

Die Ansätze des parallelen Reasonings wiederum konzentrieren sich einerseits auf die Verteilung der Arbeitslast auf eine Vielzahl an Rechenknoten im Cluster, meist über MapReduce, oder verwenden andererseits das von Mehrkern-CPUs unterstützte Multithreading. Während vor allem MapReduce basierte Ansätze (siehe z.B. WebPie [UKM+12]) die Verarbeitung von bis zu 100 Milliarden Tripeln ermöglichen und sich über weitere Hardware skalieren lassen, ist bei diesen Ansätzen gleichzeitig auch der Hardware-Einsatz und damit auch die Komplexität für die Inbetriebnahme am höchsten. Des Weiteren sind die jeweiligen Implementierungen für spezielle Regeln bzw. Ontolgiesprachen ausgelegt, auf denen aufbauend Optimierungen angewandt werden. Somit ist für jeden Regelsatz eine dedizierte Implementierung notwendig und benutzer- oder anwendungsspezifische Regeln werden nur im Rahmen des hohen Aufwands der Anpassung der Implementierung unterstützt.

Ähnliche Optimierungen werden auch bei der Parallelisierung durch Nutzung der Mehrkernprozessoren einzelner Rechenknoten angewandt. Diese sind jeweils speziell für eine Semantik implementiert, auf der aufbauend eine parallele Verarbeitung ermöglicht wird. Ein Großteil der Arbeiten nutzt dabei lediglich die parallele Ausführung auf der CPU. Eine tatsächliche Nutzung der massiv parallelen Architektur von GPUs wurde bisher nur in [HP12] und [LUQ14] gezeigt bzw. skizziert. Während in [LUQ14] das Stream-Reasoning für die ρdf Semantik adressiert wird und bisher nur die grundlegenden Ideen in Form eines Posters veröffentlicht wurden, unterliegt der in [HP12] beschriebene Ansatz der Beschränkung, dass sich nur Datensätze verarbeiten lassen, die in den Hauptspeicher der GPU geladen werden können. Gleichzeitig ergibt sich daraus die Einschränkung, dass die Arbeitslast nicht auf mehrere GPUs verteilt werden kann und somit keine Skalierung der Hardware möglich ist. Demnach nutzt der Ansatz zwar einen hohen Grad der Parallelisierung, kann jedoch keine Datensätze im Sinne des Large-scale

Reasonings (die Evaluation in [HP12] zeigt eine Verarbeitung bis ca. 36 Millionen Tripel) verarbeiten.

3.3.4 *Übertragung der Erkenntnisse auf die Forschungsfragen*

Die Betrachtung der verwandten Arbeiten hat gezeigt, dass einerseits regelbasiertes Reasoning als vielversprechend für das Large-scale Reasoning angesehen wird. Andererseits findet eine Parallelisierung bisher nur auf einer Ebene statt, auf der die anzuwendende Semantik direkt in der Implementierung umgesetzt wird. Eine tatsächliche Bereitstellung der Semantik in Form von Regeln zur Laufzeit und eine darauf aufbauende Parallelisierung, wie sie in dieser Arbeit durch die formulierten Forschungsfragen angestrebt wird, findet nicht statt. Des Weiteren konnte bisher die Nutzung von massiv paralleler Hardware nur für eine beschränkte Datenmenge gezeigt werden, ohne dass Möglichkeiten zur Skalierung, z.B. durch die Verteilung der Arbeitslast auf mehrere GPUs, bestehen.

Somit konnten durch die verwandten Arbeiten bisher nur einzelne Punkte der gestellten Forschungsfrage adressiert werden, ohne jedoch ein Gesamtsystem zu beschreiben, dass aufgrund der frei definierbaren Semantik nicht nur flexibler eingesetzt werden könnte, sondern unter der Nutzung massiv paralleler Hardware auch für mehrere Milliarden Statements ein effizientes Reasoning ermöglicht. Um diese Zielsetzung zu verfolgen, werden im weiteren Verlauf der Arbeit zunächst der Aufbau und die Funktionsweise von Productions Systems im Detail dargelegt (Kapitel 4), um in Kapitel 5 die bereits existierenden Ansätze der Parallelisierung von Production Systems (ohne einen speziellen Fokus auf das Reasoning) zu betrachten. Kapitel 6 führt in die Grundlagen der parallelen Programmierung für massiv parallele Hardware ein, um in Kapitel 7 gezielt ein Konzept der Parallelisierung des verwendeten Algorithmus zur Implementierung von Production Systems im Hinblick auf die Zielumgebung zu entwickeln. Dieser Teil der Arbeit adressiert somit die ersten beiden Detail-Forschungsfragen, die eine Nutzung moderner Mehrkernprozessoren für einen regelbasierten Reasoning-Prozess vorsehen (siehe Kapitel 1.2).

Die dritte Detail-Frage spricht insbesondere das Large-scale Reasoning und damit den Ressourcenverbrauch an. Dieser Aspekt findet aufbauend auf Kapitel 7 in Kapitel 8 und 9 Berücksichtigung. Kapitel 8 führt Konzepte für die Partitionierung und Verteilung der Arbeitslast während der Ausführung des regelbasierten Reasonings ein, um die Grundlage zur Verarbeitung großer Datenmengen auf massiv paralleler Hardware mit einem beschränkten Speicher zu schaffen. Anschließend wird in Kapitel 9 der Ressourcenverbrauch des so entstehenden Reasoning-Prozesses betrachtet, um durch Methoden der Datenkomprimierung und -strukturierung die Ressourceneffizienz zu optimieren.

4 PRODUCTION SYSTEMS ALGORITHMEN

Nach der Einführung in semantische Technologien und einem Überblick über verwandte Arbeiten im Bereich des regelbasierten und parallelen Reasonings werden im Folgenden zunächst Algorithmen zur Realisierung von Production Systems vorgestellt. Production Systems liegen immer eine Menge von Productions bzw. Regeln zugrunde, die das Verhalten eines Systems beschreiben [New73]. Jede Production besteht wiederum aus einer Menge von Pattern-Elementen, die zusammen die Left-Hand-Side (LHS) der Production bilden. Die Right-Hand-Side (RHS) ist definiert durch eine Menge von Aktionen, die ausgeführt werden, wenn die LHS erfolgreich gegen die Datenbasis evaluiert wird [Sto84]. Somit stellt die Idee der Production Systems einen wesentlichen Teil der angestrebten Reasoner-Architektur dar, die durch die beschriebene Art der Definition von Ableitungsregeln die Materialisierung von implizitem Wissen ermöglichen soll.

Ein effizienter Algorithmus zur Implementierung von Production Systems und somit zur Ausführung von Regeln wurde von Charles Forgy im Rahmen seiner Doktorarbeit [For79] entwickelt und 1982 zusätzlich unter dem Titel *„Rete: A Fast Algorithm for the Many Patterns/Many Objects Pattern Match Problem"* in *Artificial Intelligence* veröffentlicht [For82]. Wie der Titel bereits verdeutlicht, handelt es sich um einen Algorithmus, der Fakten auf bestimmte Muster, die sich beispielsweise aus Regeln ableiten lassen, überprüft bzw. *matcht*. Ausgehend von dem ursprünglichen RETE-Algorithmus existieren diverse Abwandlungen und Optimierungen, die verschiedene Aspekte wie den Speicherverbrauch oder die Optimierung durch Reduzierung der notwendigen Berechnungen adressieren. Die grundlegenden Prinzipien finden sich jedoch auch in Arbeiten wie TREAT [Mir87] wieder, einem Algorithmus, der vor allem den hohen Speicherverbrauch und den hohen Aufwand bei der Unterstützung von Negations-Ausdrücken in der Regelbasis des RETE-Algorithmus adressiert und häufig als Alternative zum RETE-Algorithmus aufgeführt wird [WH92] [NGR93].

Der hohe Verbreitungsgrad des als de-facto Standards [VW10] angesehenen RETE-Algorithmus konnte bereits in Rahmen der Betrachtung der verwandten Arbeiten gezeigt werden. Alle unter dem Kapitel 3.3.1 zusammengefassten Arbeiten implementieren den RETE-Algorithmus bzw. nutzen ein System, dem dieser zugrunde liegt. Entsprechend kann bereits festgehalten werden, dass sich der Algorithmus grundlegend zur Anwendung monotoner Semantik für das Ableiten von implizitem Wissen auf RDF Daten eignet und sich alternative Algorithmen

zur Implementierung von Production Systems aus dem RETE-Algorithmus ableiten. Aus diesem Grund findet im Folgenden eine detaillierte Betrachtung der Funktionsweise des Algorithmus statt, die der weiteren Arbeit als Grundlage für die Überführung eines Production Systems Algorithmus auf eine massiv parallele Ausführung dient.

4.1 REGELN UND REGEL-SYNTAX

Regeln werden im Allgemeinen in der Form *If-Then* aufgestellt und bestehen aus einem *Body* (dem *If*-Teil oder auch Left-Hand Side (LHS) genannt), der eine zu erfüllende Bedingung darstellt, und einem *Head* (Right-Hand Side (RHS)), der die Konsequenzen bzw. auszuführende Aktion beschreibt.

> WENN Punktezahl > 95 DANN Note = „Sehr gut"

Die oben aufgeführte Regel drückt z.B. aus, dass ab einer Punktzahl von mindestens 96 Punkten die Note „Sehr gut" vergeben werden soll (unter der Annahme, dass es sich bei *WENN* und *DANN* um Schlüsselwörter handelt).

Regeln können durch Verknüpfung mittels logischer Operatoren auch aus mehr als einer Bedingung bestehen. Ebenso lassen sich mehrere Aktionen durch Aneinanderreihung definieren. So lässt sich das einführende Beispiel wie folgt erweitern:

> WENN Punktezahl > 95 UND Bonusaufgaben = Erledigt DANN Note = „Sehr gut", Erstelle Vermerk

In dem erweiterten Beispiel wird zusätzlich geprüft, ob Bonusaufgaben erledigt worden sind. Wenn beide Bedingungen zutreffen, wird nicht nur die Note „Sehr gut" vergeben, sondern auch ein gesonderter Vermerk erstellt.

Während die zuvor aufgeführten Regeln zur leichteren Verständlichkeit umgangssprachlich formuliert sind, wird den im weiteren Verlauf der Arbeit verwendeten Regeln eine Syntax zugrunde gelegt. Sie lässt die Definition von Forward-Chaining-Regeln zu und ist stark an die Jena[1] Regel-Syntax angelehnt[2]. Auf den Aspekt des Ausführens einer Funktion als Aktion beim Feuern einer Regel wurde ebenso verzichtet wie auf die Möglichkeit der Definition von Backward-Chaining

---

1 Jena ist ein Java-basiertes Open Source Framework zur Entwicklung semantischer Anwendungen. Es beinhaltet unter anderem eine generische Rule-Engine, die zur Anwendung verschiedener Semantik genutzt werden kann.

2 Im weiteren Verlauf der Arbeit werden Teile des Jena-Frameworks für die Implementierung der erarbeiten Reasoner-Architektur genutzt (z.B. der Regel-Parser), weshalb auch die Jena Regel-Syntax verwendet wird.

```
Rule      :=   bare-rule .
          or   [ bare-rule ]
          or   [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> term, ... term    // forward rule

term      :=   (node, node, node)           // triple pattern

node      :=   uri-ref                      // e.g. http://foo.com/eg
          or   prefix:localname             // e.g. rdf:type
          or   <uri-ref>                    // e.g. <myscheme:myuri>
          or   ?varname                     // variable
          or   'a literal'                  // a plain string literal
          or   'lex'^^typeURI               // a typed literal, xs:* type names
               supported
          or   number                       // e.g. 42 or 25.5
```

Listing 4: Definition der Regel-Syntax. In Anlehnung an [Apa14]

Regeln, da dies nicht im Fokus der Arbeit steht und das Ausführen einer Funktion als Aktion einer Ausführung auf der GPU entgegen stünde. Eine nachträgliche Unterstützung dieser Funktionalität ließe sich jedoch unter Berücksichtigung einer Sonderbehandlung (ein Aufruf beliebiger Funktionen z.B. in einem Java-Programm aus dem Ausführungskontext auf einer GPU heraus ist nicht möglich) entsprechender Regeln leicht realisieren.

Entsprechend der in Listing 4 definierten Syntax können Regeln optional in eckigen Klammern eingeschlossen sein und einen Namen besitzen. Weiterhin können sie aus einer Aneinanderreihung beliebig vieler Regel-*Terme* (ein Term bildet jeweils eine Bedingung ab) bestehen, die zusammen den Body bilden. Der Body ist vom Head der Regel, also der Konsequenz, die beim Feuern einer Regel ausgeführt wird, durch die Zeichenkette „->“ getrennt. Sowohl Body- als auch Header-Terme bestehen aus jeweils drei *Nodes* (entspricht einem Tripel), die im Wesentlichen aus einer URI, einer Variablen oder einem Literal bestehen können. Diese Syntax reicht bereits aus, um beispielsweise die RDFS- oder pD* Semantik durch Ableitungsregeln auszudrücken. Eine beispielhafte Darstellung von zwei Regeln ist in Kapitel 4.2.1 dargestellt.

4.2 RETE-ALGORITHMUS

Basierend auf einem gegebenen Regelsatz und einer Faktenbasis, die das vorhandene Wissen repräsentiert, führt der RETE-Algorithmus ein *Pattern-Matching* aus, in dem alle Fakten gegen das Muster der einzelnen Bedingungen geprüft werden,

um so gültige Faktenkombinationen zu finden. Eine gültige Faktenkombination kann als *Match* einer Regel betrachtet werden und bildet die Grundlage für das Feuern einer Regel. Durch das Feuern werden neue Fakten erzeugt, die wiederum als Ausgangsbasis für eine erneute Iteration des Algorithmus dienen. Dieser Vorgang wiederholt sich, bis keine neuen Fakten mehr abgeleitet werden können. Die folgenden Abschnitte erläutern die dafür notwendigen Schritte im Detail und bauen eine Verständnisgrundlage für die spätere Parallelisierung des Algorithmus auf. Aufgrund der Fokussierung dieser Arbeit auf die Betrachtung monotoner Regeln, die die Grundlage für weit verbreitete Regelsätze insbesondere für Large-scale Reasoning bilden [BG00] [tH05] [MPG07], beschränkt sich die Einführung des RETE-Algorithmus auf die für diese Arbeit relevanten Aspekte. Entsprechend findet keine Betrachtung von negierten Regelausdrücken bzw. negativen Bedingungen statt, die beispielsweise dazu führen können, dass ein zuvor berechnetes Ergebnis aufgrund neu abgeleiteter Fakten seine Gültigkeit verliert.

4.2.1 *RETE-Netz*

Die Einführung in den RETE-Algorithmus soll anhand eines konkreten Beispiels erfolgen, das bereits in [PBSZ13b] vorgestellt wurde. Um schon im Beispiel den Bezug zu RDF-Daten und dem Reasoning herzustellen, werden zwei Regeln aus der RDFS Semantik als Ausgangsbasis genutzt. Die unten aufgeführte Regel R_1 besteht aus nur einem Regel-Term, der erfüllt sein muss, um die Regel zu feuern. R_2 hingegen verknüpft zwei Regel-Terme, weshalb für eine Auswertung genau zwei Fakten verknüpft werden müssen. Variablen in den Regeln sind durch ein „?“ gekennzeichnet. Sie markieren die Teile einer Regel, die variabel durch Ausdrücke aus der Faktenbasis belegt werden können.

$$(?x\ \ ?p\ \ ?y) \rightarrow (?p\ \ \text{rdf:type}\ \ \text{rdf:Property}) \tag{R_1}$$

$$(?x\ \ ?p\ \ ?y),\ \ (?p\ \ \text{rdfs:domain}\ \ ?c) \rightarrow (?x\ \ \text{rdf:type}\ \ ?c) \tag{R_2}$$

Um den Pattern-Matching-Prozess effektiv zu gestalten, wird als erster Schritt des RETE-Algorithmus aus der gegebenen Regelbasis ein Netz aus sogenannten *Alpha* und *Beta* Knoten erstellt. Für jeden individuellen Term wird dabei ein Alpha-Knoten erstellt. Sich wiederholende Regel-Terme, wie z.B. $(?x\ \ ?p\ \ ?y)$, der sowohl in R_1 als auch in R_2 vorkommt, werden auf einen einzigen Alpha-Knoten abgebildet. Im Gegensatz zu Alpha-Knoten, die keine Eltern-Knoten besitzen, haben Beta-Knoten grundsätzlich zwei Eltern-Knoten. Sie drücken die Verbindung von zwei oder mehr Regel-Termen einer Regel aus. Entsprechend können die Eltern-Knoten sowohl aus Alpha als auch aus Beta-Knoten bestehen. Durch Anwenden

der beschriebenen Vorgehensweise ergibt sich aus den Regeln R_1 und R_2 das in Abbildung 3 abgebildete RETE-Netz.

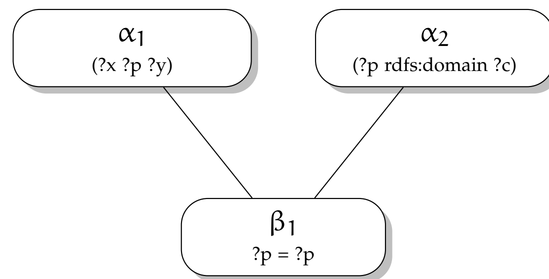


Abbildung 3: RETE-Netz der Regeln R_1 und R_2

Aus dem gegebenen Regelsatz leiten sich insgesamt zwei Alpha-Knoten ab, die den zwei unterschiedlichen Regel-Termen entsprechen (der Term $(?x\ ?p\ ?y)$ kommt sowohl in R_1 als auch in R_2 vor und wird auf einen einzigen Knoten im Netz abgebildet). Um die Verknüpfung der beiden Regel-Terme aus R_2 zu berücksichtigen, wird zusätzlich ein Beta-Knoten erstellt. Dieser verbindet α_1 und α_2 und bildet somit die Regel R_2 vollständig ab, während Regel R_1 bereits durch α_1 komplett beschrieben ist. Für Regeln mit mehr als zwei Regel-Termen können sich zusätzlich Beta-Knoten bilden, deren Eltern-Knoten selber auch Beta-Knoten sind und zu einer höheren *Tiefe* des Netzes beitragen.

Durch die beschriebene Vorgehensweise zum Ableiten der Alpha- und Beta-Knoten ergibt sich ein RETE-Netz, das in Abhängigkeit des Regelsatzes an Komplexität und Tiefe variieren kann und die Ausgangsbasis für das Pattern-Matching darstellt.

4.2.2 *Pattern-Matching*

Das aus der Regelbasis abgeleitete Netz dient als Ausgangsbasis für den eigentlichen Pattern-Matching-Prozess. Dieser beginnt mit dem *Alpha-Matching*, bei dem sämtliche Fakten aus der Faktenbasis mit jedem Alpha-Knoten *gematched*, also auf Übereinstimmung mit dem vorgegebenen Muster geprüft werden. Das Muster definiert sich durch die nicht variablen Teile des entsprechenden Regel-Terms, der den Ursprung eines Alpha-Knotens bildet. Beispielsweise stimmen alle Tripel, die als Prädikat ein *rdfs:domain* besitzen, mit dem Muster von α_2 überein. Um die-

sen Prozess zu verdeutlichen, wird im Folgenden eine Faktenbasis mit den drei Tripeln T_1, T_2 und T_3 eingeführt[3]:

$$< \text{Bob} \;\; \text{uni:publishes} \;\; \text{Paper1} > \qquad (T_1)$$

$$< \text{Alice} \;\; \text{uni:publishes} \;\; \text{Paper2} > \qquad (T_2)$$

$$< \text{uni:publishes} \;\; \text{rdfs:domain} \;\; \text{Researcher} > \qquad (T_3)$$

Jedes der vorliegenden Tripel wird auf Übereinstimmung mit dem Muster aller Alpha-Knoten geprüft. Eine Besonderheit bildet dabei α_1, da das Muster aus drei Variablen besteht und somit keine Bedingung enthält. Entsprechend stimmen grundsätzlich alle Tripel mit dem Muster überein und können als *Match* für α_1 bezeichnet werden. Basierend auf den gefundenen Übereinstimmungen wird für jeden Knoten ein *Working-Memory* angelegt, in dem je nach Implementierung jeweils eine Referenz oder eine Kopie der Tripel abgelegt wird.

Im Anschluss an das Alpha-Matching folgt das Beta-Matching, das auf die Working-Memories der Alpha-Knoten zugreift und somit zeitlich vom Alpha-Matching abhängig ist. Beim Beta-Matching wird jedes Element aus dem Working-Memory des ersten Eltern-Knotens mit jedem Element des Working-Memories des zweiten Eltern-Knotens zusammengeführt, um zu überprüfen, ob die Kombination dem Muster des jeweiligen Beta-Knotens entspricht. Die Übereinstimmung der nicht variablen Teile ist dabei bereits durch das Alpha-Matching sichergestellt, so dass beim Beta-Matching die in mehreren Regel-Termen auftretenden variablen Teile auf Übereinstimmung geprüft werden müssen. Beispielsweise stellt die Variable ?p in Regel R_2 einerseits das Prädikat des ersten Regel-Terms und andererseits auch das Subjekt des zweiten Regel-Terms dar. Entsprechend müssen das Prädikat eines Eintrags aus dem Working-Memory von α_1 und das Subjekt eines Eintrags aus α_2 identisch sein, um als Match für β_1 identifiziert zu werden. Das Working-Memory für Beta-Knoten setzt sich demnach aus einer Kombination von Tripeln bzw. Tripel-Referenzen zusammen.

Abbildung 4 zeigt das RETE-Netz, nachdem die gegebene Faktenbasis durch das Netz propagiert worden ist. Das Working-Memory von α_1 beinhaltet entsprechend der Faktenbasis mit drei Tripeln drei Einträge (aus Gründen der Übersichtlichkeit werden in Abbildung 4 nicht nur die Referenzen T_1, T_2, T_3 angegeben, sondern auch die eigentlichen Tripel). Die Bedingung von α_2 wird hingegen nur von T_3 erfüllt. Für β_1 können zwei gültige Kombinationen gebildet werden, die aus T_1 und T_3 bzw. T_2 und T_3 bestehen.

3 Das Präfix *uni:* wird genutzt, um die URI einer beispielhaften Ontologie abzukürzen, die Konzepte eines Universitäts-Szenarios beschreibt. *rdfs:* hingegen referenziert den RDF-Schema Namensraum http://www.w3.org/2000/01/rdf-schema#.

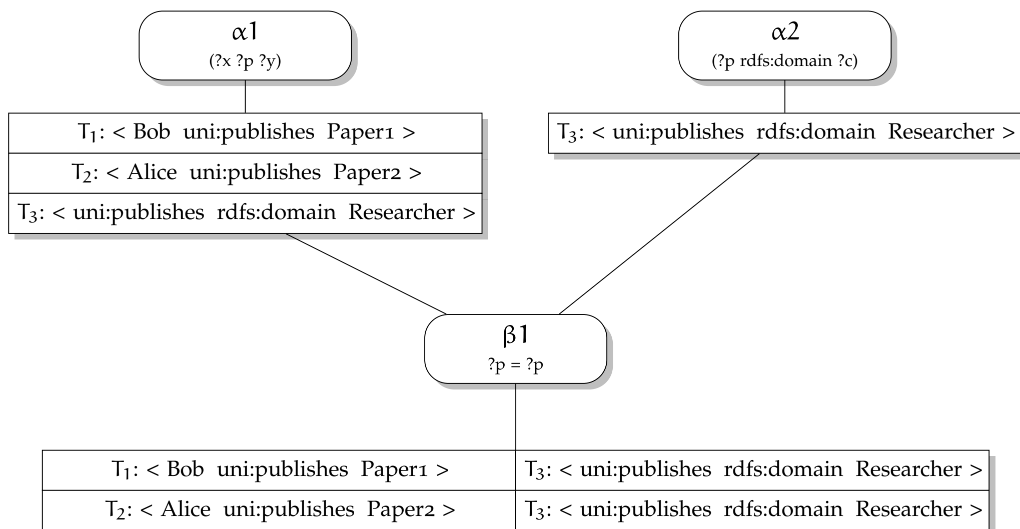


Abbildung 4: RETE-Netz mit Working-Memories

4.2.3 *Regelausführung*

Nachdem der Matching-Prozess für Beta-Knoten abgeschlossen ist, folgt das eigentliche Ausführen der Regeln (auch *feuern* genannt). Dazu werden die Working-Memories der jeweiligen *Terminal-Knoten* genutzt (Terminal-Knoten bezeichnen jeweils den Knoten im RETE-Netz, der eine Regel vollständig abbildet), um neue Fakten nach der Vorschrift der RHS einer Regel zu erstellen. Entsprechend dient das Working-Memory von α_1 zum Feuern der Regel R_1 und das Working-Memory von β_1 zum Feuern von R_2. Als Resultat ergeben sich vier neue Fakten und ein Duplikat (Tripel T_4 wird zweimal abgeleitet, aber nur einmal dazu genutzt, um die bestehende Faktenbasis zu erweitern), die im Folgenden aufgelistet sind:

< uni:publishes rdf:type rdf:Property > (T_4)

< rdfs:domain rdf:type rdf:Property > (T_5)

< Bob rdf:type Researcher > (T_6)

< Alice rdf:type Researcher > (T_7)

Der Prozess des Erkennens und dem darauf aufbauenden Verwerfen von Duplikaten wird im Folgenden auch *Deduplikation* genannt und stellt sicher, dass keine expliziten Informationen mehrfach für einem Datenbestand abgeleitet und gespeichert werden. Die neu abgeleiteten Fakten werden ebenfalls durch das RETE-Netz propagiert und können so zur Ableitung weiterer Fakten beitragen. Dieser Pro-

zess wiederholt sich, bis keine neuen Aussagen mehr entstehen und damit der RETE-Algorithmus terminiert.

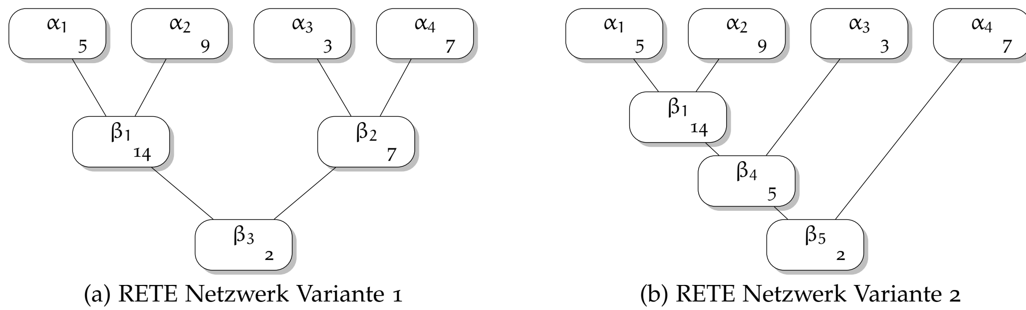
Die Stärke des RETE-Algorithmus besteht darin, dass die Working-Memories den Zustand des RETE-Netzes über die gesamte Ausführungszeit hinweg speichern und somit die Komplexität des Matching-Prozesses ab der zweiten Iteration lediglich von der Anzahl der in der vorherigen Iteration neu abgeleiteten Fakten abhängt. Der während der ersten Iteration häufig rund 90% [Gup84] der Ausführungszeit einnehmende Match-Prozess kann so für alle weiteren Iterationen drastisch verkürzt werden, was zu einer schnelleren Ausführung bei einem gleichzeitig erhöhtem Speicherbedarf führt.

4.3 OPTIMIERUNGEN UND ERWEITERUNGEN DES RETE-ALGORITHMUS

Nach der Einführung in die grundlegende Funktionsweise des RETE-Algorithmus wird im Folgenden ein Überblick über mögliche Erweiterungen und Optimierungen gegeben, die seit der Einführung im Jahr 1979 [For79] zur Adressierung verschiedener Herausforderungen veröffentlicht wurden. Der Überblick erhebt keinen Anspruch auf Vollständigkeit, sondern dient der zusätzlichen Verständnisbildung und gibt die Möglichkeit, bestehende Ansätze im weiteren Verlauf der Arbeit zu berücksichtigen. Die Betrachtung von Parallelisierungsansätzen von Production Systems sowie des RETE-Algorithmus im Speziellen erfolgt im Anschluss in Kapitel 5.

4.3.1 *Optimierungen basierend auf der Netzwerkgestaltung*

Ein Nachteil des RETE-Algorithmus liegt in der potentiell hohen Anzahl an Kombinationen einzelner Datenelemente, die während des Beta-Matching berechnet werden müssen bzw. anschließend im Working-Memory der Beta-Knoten gespeichert werden. Beispielsweise kann eine nicht sorgfältig gewählte Regelbasis bei einem einfachen RETE-Netzwerk mit nur einem Beta-Knoten bereits für n Eingabedaten zu n^2 Einträgen im Beta Working-Memory führen. Die Gesamtzahl solcher Kombinationen kann bei einem komplexeren RETE-Netzwerk mit Beta-Knoten auf mehreren Ebenen zu einer Polynomfunktion höheren Grades anwachsen und so mit einer wachsenden Anzahl an Datenelementen zu einer kombinatorischen Explosion führen [AT93]. Ausschlaggebend für die entstehende Komplexität ist die Regelbasis bzw. die Struktur des aus der Regelbasis abgeleiteten Netzwerks. Dieses lässt sich aufgrund der Assoziativität sowie der Kommutativität der Regel-Terme [Ish88] auf verschiedene Weise aufbauen. Abbildung 5 zeigt beispielsweise zwei mögliche Netzwerke, die sich aus einer einzigen Regel, bestehend aus vier Regel-Termen (abgebildet auf α_1, α_2, α_3 und α_4), ableiten lassen. Innerhalb der



(a) RETE Netzwerk Variante 1 (b) RETE Netzwerk Variante 2

Abbildung 5: Verschiedene Möglichkeiten der Netzwerk-Bildung

Knoten ist in den abgebildeten Netzwerken die jeweilige Anzahl an Matches für einen beispielhaften Datensatz notiert. Basierend auf diesen Annahmen ergibt sich für das Netzwerk (a) eine Anzahl an notwendigen Beta-Match Operationen (*Joins*) von insgesamt 164. Diese errechnet sich wie folgt:

$$(5 * 9) + (3 * 7) + (14 * 7) = 164$$

Für das RETE-Netzwerk (b) hingegen, das ein identisches Resultat liefert, aber eine andere Struktur aufweist, ergibt sich die Anzahl der notwendigen Join-Operationen aus:

$$(5 * 9) + (14 * 3) + (5 * 7) = 122$$

Während sowohl die Ausgangsbasis als auch das Endergebnis (das Working-Memory für β_3 bzw. β_5) in beiden Netzwerken identisch sind, mussten für das Netz (b) nur rund 25% weniger Join-Operationen ausgeführt werden. Ähnlich wirkt sich die Struktur des Netzwerks auch auf die Speicherbelastung aus. Unter ausschließlicher Betrachtung der Beta-Knoten müssen für das Netzwerk (a) beispielsweise insgesamt 23 Matches gespeichert werden, während es für Netzwerk (b) lediglich 21 sind.

Unter Berücksichtigung der zuvor aufgeführten Unterschiede durch die Netzwerkgestaltung wird eine Optimierung der RETE-Ausführung, basierend auf einem möglichst optimal gestalteten Netzwerk, in verschiedenen Arbeiten mit unterschiedlichen Ansätzen verfolgt. In [Ish94] werden dazu zunächst drei Kategorien häufig verfolgter Strategien beschrieben, die im Folgenden vorgestellt werden.

Most specific first

Die *most specific first* oder auch *most restrictive first* Strategie sieht vor, dass Knoten innerhalb des Netzwerks nach ihrer Restriktion sortiert und platziert werden [WH92] [Ish88] [Ish94] [ÜÖÖ05]. Ziel ist es, zunächst Join-Operationen von Knoten zu berechnen, die z.B. durch ihre hohe Anzahl an Bedingungen (und damit einer möglichst geringen Anzahl an Variablen) eine potentiell niedrige Anzahl an Fakten im Working-Memory besitzen. Die Bemessung der jeweiligen Restriktionen kann entsprechend der Anzahl der Variablen innerhalb eines Regel-Terms oder der Anzahl der Elemente im Working-Memory erfolgen. Weitere Betrachtungen wie die Bewertung anhand des Prädikats eines Regel-Terms sind ebenfalls möglich [ÜÖÖ05] [ÖÖÜ07].

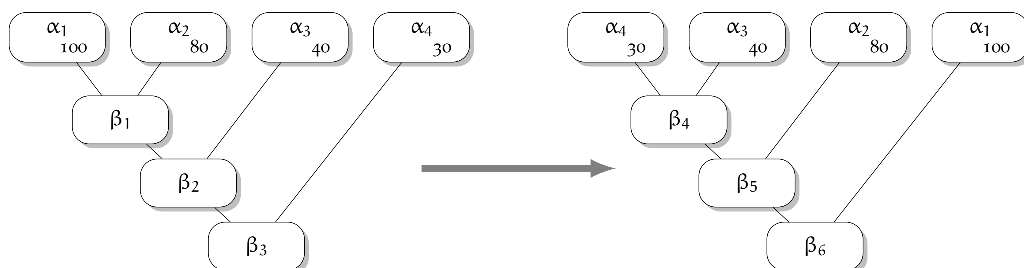


Abbildung 6: RETE-Optimierung: Most specific first. Vgl. [Ish94]

Aufbauend auf einer Bewertung der Restriktion zeigt Abbildung 6 eine entsprechende Umstrukturierung (die Restriktion wird in der Abbildung anhand der Anzahl der Einträge im Working-Memory festgestellt), die eine Umsortierung der Alpha-Knoten beinhaltet, um die Anzahl der Join-Operationen für die folgenden Beta-Knoten zu reduzieren.

Most volatile condition last

Die Strategie der *Most volatile condition last* adressiert die Tatsache, dass neue Fakten, die dem Netzwerk über Alpha-Knoten zugeführt werden, zu einer Berechnung von Join-Operationen in allen folgenden Beta-Knoten führen [Ish88] [WH92] [Ish94]. Entsprechend sind vor allem Knoten zu betrachten, die z.B. aufgrund der Ableitung von neuen Fakten häufig Änderungen erfahren. Damit, wie in der Abbildung 7 (links) aufgezeigt, diese Änderungen nicht jeweils zu Join-Berechnungen in β_1, β_2 und β_3 führen, werden die Knoten mit vielen Änderungen dem Netzwerk möglichst spät beigefügt. So müssen beispielsweise für das Netzwerk auf der rechten Seite von Abbildung 7 für neue Fakten in α_2 nur Join-Operationen in

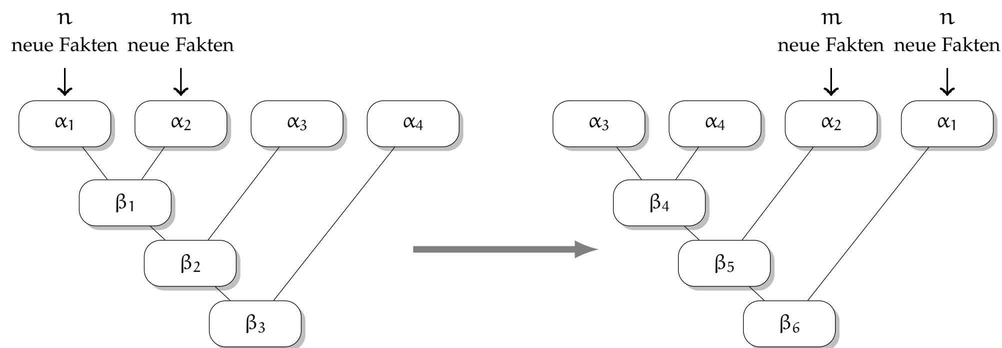


Abbildung 7: RETE-Optimierung: Most volatile condition last, mit $n > m$. Vgl. [Ish94]

β_5 und β_6 ausgeführt werden, während sich neue Fakten in α_1 ausschließlich auf β_6 auswirken.

Share join clusters among rules

Neben der Umstrukturierung des RETE-Netzwerks kann vor allem die Wiederverwendung einzelner Knoten oder Teilgraphen die Ausführungsgeschwindigkeit des Algorithmus maßgeblich beeinflussen. Diese Strategie wird in [Ish94] als *Share join clusters among rules* bezeichnet und basiert auf der Tatsache, dass sich einzelne Regel-Terme häufig in verschiedenen Regeln wiederholen. Durch eine Zusammenführung der Netze aller Regeln kann die mehrfache Berechnung von Ergebnissen vermieden und stattdessen eine Wiederverwendung eingeführt werden. Auf diese Weise kann alleine die gemeinsame Nutzung eines Alpha-Knotens zu einer Zeit- und Speicherersparnis von $1/n$ führen, wenn ein Alpha-Knoten von n Regeln wiederverwendet wird [TKO11]. Abbildung 8 verdeutlicht die Maßnahme der Zusammenführung gemeinsamer Teilgraphen, bei der zwei unabhängige Netzwerke vereint werden, um die Knoten α_1, α_2 und β_1 gemeinsam zu nutzen. Die Anwendung der Strategie der Wiederverwendung von Teilgraphen innerhalb des RETE-Netzwerks wird u.a. auch in [Ish88], [Mad03], [YYW11] und [TKO11] aufgeführt und wird im Folgenden als *Node-Sharing* referenziert.

Aufgrund des Konfliktpotentials, dass zwischen den verschiedenen Ansätzen der Optimierung besteht [Ish88] [Ish94] (das von der *most specific first* Strategie berechnete Netzwerk kann z.B. anders gestaltet sein als das von der *most volatile condition last* Strategie), wird in [Ish88] eine hybride Vorgehensweise zur RETE-Netz-Entwicklung vorgeschlagen, die zunächst, basierend auf einfachen Ausführungszyklen, ein Kostenmodell für die einzelnen Knoten bildet. Optimierungen der Knoten mit den höchsten Kosten werden anschließend am stärksten priorisiert, während Optimierungen für Knoten mit weniger hohen Kosten ggf. nicht voll-

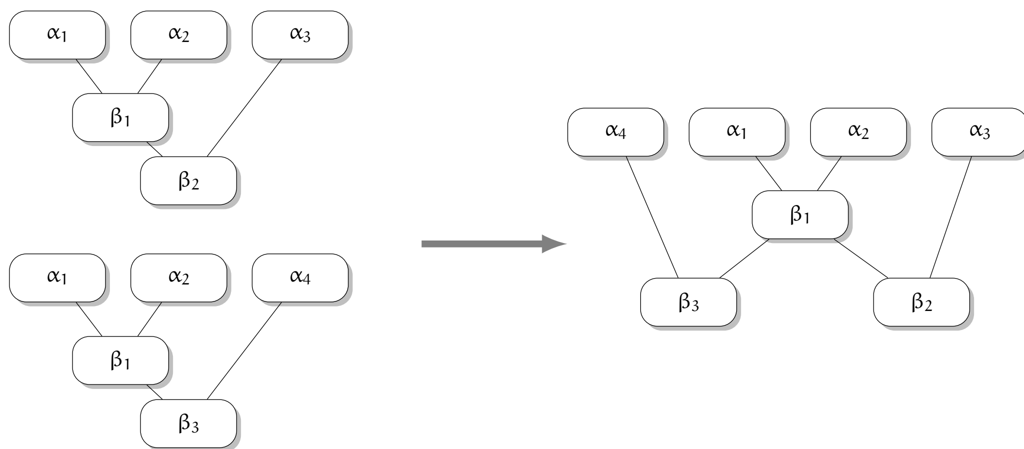


Abbildung 8: RETE-Optimierung: Share join clusters among rules.

ständig berücksichtigt werden können. Ein ähnliches Vorgehen wird in [ÖÖÜ07] vorgeschlagen, indem die *most specific first* Heuristik mit einer Strategie kombiniert wird, die versucht, Regel-Terme mit gemeinsamen Variablen über einen Beta-Knoten direkt miteinander zu verbinden.

Zusätzlich zu den bereits vorgestellten Optimierungen wird in [TKO11] und [TKO13] eine in zwei Phasen geteilte RETE-Ausführung vorgestellt, die eine datenspezifische Nutzung der *Most specific first* Strategie ermöglicht. Dazu werden zunächst nur die Alpha-Knoten erstellt (unter Berücksichtigung der Wiederverwendung von Knoten) und das Alpha-Matching ausgeführt. Aufbauend auf den so gesammelten Daten über die Anzahl der jeweiligen Elemente in den Working-Memories erfolgt eine Erstellung der Beta-Knoten sowie die weitere Ausführung des RETE-Algorithmus. Ebenso berücksichtigt die vorgestellte Arbeit die *Connectivity*-Heuristik, die sicherstellt, dass zusammengeführte Knoten mindestens eine Variable gemeinsam haben [TKO11]. Des Weiteren wird in den Arbeiten [TKO11] und [TKO13] ein Algorithmus zur Auswahl der anzuwendenden Regeln basierend auf der genutzten Ontologie vorgestellt. Dazu wird zunächst ein Vergleich der in einem Datenbestand vorhandenen OWL-Konstrukte mit den in der Regelbasis vorhandenen Regeln durchgeführt. Aufbauend darauf werden nur die Regeln für den Reasoning-Vorgang ausgewählt, die tatsächlich zur Materialisierung von implizitem Wissen führen können.

4.3.2 *Optimierungen basierend auf einer erweiterten Behandlung von RETE-Knoten*

Neben der optimalen Strukturierung eines RETE-Netzwerks zur Reduzierung der Join-Berechnungen wird in [AT93] eine weitere Möglichkeit vorgestellt, den Prozess des Beta-Matching zu beschleunigen. Dazu wird ein mengenorientiertes Ver-

fahren (Collection oriented) vorgeschlagen, das statt der Iteration einzelner Fakten die Iteration von Fakten in zusammenhängenden Mengen durch das RETE-Netzwerk vorsieht. Durch die Betrachtung von Mengen während der Join-Operationen soll die Anzahl der kombinatorischen Möglichkeiten drastisch reduziert werden, was neben einer verringerten Anzahl an Berechnungen gleichzeitig auch zu einer Verringerung der Speicherbelastung führt [AT93]. Das gleiche Ziel der Reduzierung der notwendigen Berechnungen wird in [TNR90] durch die Einschränkung der Expressivität der Regelausdrücke verfolgt. Dazu werden Unique-Attribute eingeführt, die die Anzahl der Einträge in Beta-Memories auf einen einzigen begrenzen. Aufbauend auf der Verwendung von Unique-Attributen wird in [TKR92] der Uni-RETE-Algorithmus eingeführt, der zusätzlich die Speicherbelastung für Beta-Memories minimiert, indem keine vollständige Abbildung der Daten in den Beta-Memories mehr vorgehalten wird. Stattdessen lassen sich diese aus den Einträgen der vorherigen Beta-Memories ableiten.

Eine Optimierung des Match-Prozesses wird auch in [DWH09] vorgeschlagen. Dabei wird für das Alpha-Matching ein Hash-basiertes Verfahren eingeführt, das eine direkte Zuordnung von Fakten zu Alpha-Knoten erlaubt. Um die Arbeitslast für das Beta-Matching zu reduzieren, werden Beta-Knoten durch zusätzlich eingeführte Knoten gruppiert. Bevor Fakten einen Gruppierungsknoten durchlaufen, werden bereits erste Bedingungen überprüft, die auch für die darunter liegenden Beta-Knoten erfüllt sein müssen. Nur wenn die Überprüfung positiv verläuft, werden die Fakten auch an die jeweils gruppierten Beta-Knoten weitergereicht. Eine ähnliche Strategie wird auch in [XZ10] vorgestellt. Im Gegensatz zu [DWH09] wird in [XZ10] jedoch auch für das Beta-Matching eine Hash-Methode verwendet, um Fakten für eine Join-Operation möglichst effizient identifizieren zu können.

Die Verwendung von Gruppierungsknoten wird auch für den IRETE-Algorithmus [YYW11] unter der Bezeichnung *ObjectTypeNode* eingeführt. Ebenso wie die Gruppierungsknoten in [DWH09] erfüllen sie die Aufgabe der frühzeitigen Filterung und anschließenden Weiterverteilung von Fakten. Für die effiziente Verteilung von Fakten nach der Typ-basierten Filterung an Alpha-Knoten wird wie in [DWH09] und [XZ10] ein Hash-basiertes Verfahren angewandt.

Weitere Optimierungen, die auf einer erweiterten Verwendung des RETE-Netzes basieren, existieren beispielsweise für spezielle Anwendungsbereiche. In [GQH13] und [KLK+14] wird die Nutzung von Regeln speziell für kontextbasierte Anwendungen unter Einbezug des RETE-Algorithmus adressiert. Die in [GQH13] vorgestellte Arbeit führt ein zusätzliches Caching zur effizienteren Berechnung des Alpha-Matchings ein, dem die Annahme zugrunde liegt, dass für kontextbezogene Berechnungen, z.B. durch eine Wertänderung, häufig die gleichen Regeln erneut ausgeführt werden. In [KLK+14] stellen die Autoren den RETE-ADH Algorithmus (der Name steht für *RETE-Alpha network Dual Hashing Algorithmus*) vor,

der ein zusätzliches Hashing einführt, um auch die Fakten innerhalb der Working-Memories von Alpha-Knoten zu strukturieren.

4.3.3 *Zusätzliche Optimierungen und Erweiterungen*

Der zu Beginn des Kapitels bereits erwähnte TREAT-Algorithmus [Mir87] stellt eine Abwandlung des RETE-Algorithmus dar und wird in verschiedenen Arbeiten für einen Vergleich der Ausführungsgeschwindigkeit genutzt [WH92] [NGR93]. Statt der Pflege von Beta-Memories, die besonders aufwändig sein kann, wenn die verwendete Syntax Regelausdrücke mit negierten Ausdrücken zulässt, werden die Beta-Matches für jede Iteration des Algorithmus neu berechnet. Der tatsächliche Performancegewinn oder -verlust kann jedoch stark durch den verwendeten Regelsatz sowie den zu verarbeitenden Datensatz variieren. So zeigt eine Evaluation in [WH92] beispielsweise die Überlegenheit von TREAT, während in [NGR88] RETE als der in vielen Fällen effizientere Algorithmus beschrieben wird. Die bereits vorgestellten Optimierungen lassen sich entsprechend der identischen Konzepte (abgesehen von der Pflege der Beta-Memories) auf beide Algorithmen anwenden.

Einige Arbeiten zur Erweiterungen des RETE-Algorithmus für die Behandlung von Unsicherheiten wurden bereits während der Vorstellung der verwandten Arbeiten des parallelen Reasonings in Kapitel 3.3 erwähnt [LQWY11] [LQWY12] [ZQL+12]. [SMP08] und [SMP10] ergänzen diese Angaben und führen entsprechende Konzepte ein, die entgegen der zuvor genannten Arbeiten nicht für eine parallele bzw. verteilte Ausführung des Algorithmus ausgelegt sind.

5 PARALLELISIERUNG VON PRODUCTION SYSTEMS

Im vorherigen Kapitel 4 wurden bereits Production Systems Algorithmen und im speziellen der RETE-Algorithmus vorgestellt und die verwandten Arbeiten im Bereich der Optimierungen und Erweiterungen aufgezeigt. In diesem Kapitel erfolgt die Betrachtung der verwandten Arbeiten mit dem Fokus auf die Parallelisierung von Production Systems. Dabei wird eine Untergliederung in *Regel-Partitionierung* und *Daten-Partitionierung* vorgenommen. Unter der Regel- Partitionierung werden in dieser Arbeit Ansätze verstanden, die direkt auf den gegebenen Regeln aufsetzen oder die aus den Regeln abgeleitete Netzwerkstruktur zur Parallelisierung nutzen. Dem gegenüber steht die Daten-Partitionierung, bei der die zu verarbeitenden Daten als Ausgangsbasis für eine Parallelisierung dienen.

5.1 REGEL-PARTITIONIERUNG

Erste Arbeiten zur Parallelisierung von Production Systems wurden bereits zu der Zeit der Veröffentlichung des RETE-Algorithmus 1979 bzw. 1982 publiziert [SS82] [Sto84] [FGW84] [IS85] [GFNW86] [Gup86]. In [SS82] und [Sto84] beschreiben die Autoren die parallele Multi-Prozessor Architektur DADO, die zu einer signifikanten Performancesteigerung bei der Ausführung von Production Systems führen soll. Die aus bis zu mehreren tausend Prozessoren bestehende Architektur ist als vollständiger Binärbaum aufgebaut und sieht für jeden Prozessor einen lokalen Speicher vor. Der Binärbaum wird logisch unterteilt in Upper Tree, PM-Level und WM Subtrees (siehe Abbildung 9). Während der Upper Tree zur Datenbereitstel-

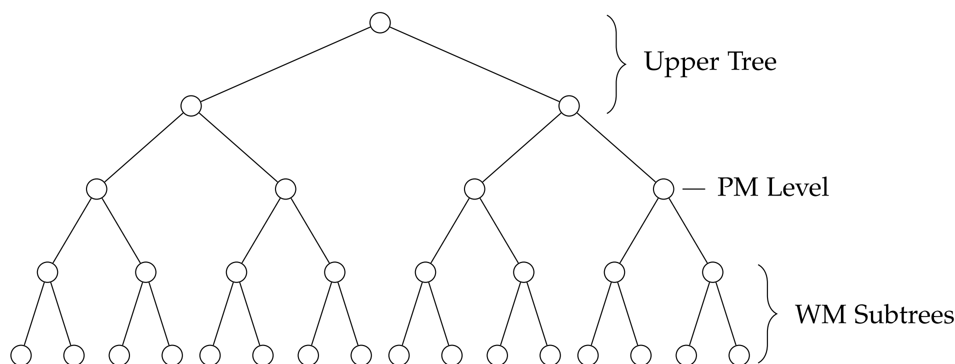


Abbildung 9: DADO Prozessorarchitektur. Vgl. [SS82]

lung und Synchronisation genutzt wird, dienen die Prozessoren des PM Level für das direkte Mapping auf Productions. Entsprechend wird jeder Production exakt ein Prozessor des PM Level zugewiesen. Auf diese Weise wird bereits eine Parallelisierung auf der Ebene der Production erreicht. Eine weitere Parallelisierung wird durch den zum Teil parallel ausgeführten Match-Prozess basierend auf den WM Subtrees realisiert. Dazu wird von den PM-Knoten eine Broadcast-Nachricht mit Match-Informationen an die jeweiligen WM Subtrees gesendet. Alle Subtrees, deren Evaluation der gegebenen Bedingungen nicht positiv ausfällt, werden für die weiteren Operationen deaktiviert.

Im Gegensatz zu Stolfo et al. [SS82] [Sto84] betrachten Gupta et. al in [FGW84] die Möglichkeiten der Parallelisierung nicht an einer konkreten Hardware, sondern stellen vielmehr Anforderungen an eine entsprechende Prozessorarchitektur auf. Dabei führen sie verschiedene Möglichkeiten der Parallelisierung ein, die als *Production-Level*, *Condition-Level* und *Action-Level* klassifiziert werden. Production-Level-Parallelisierung wird so verstanden, dass die Productions gruppiert und einem Prozessor zugewiesen werden. Als maximale Ausprägung dieser Strategie ergibt sich entsprechend die Verteilung von je einer Production auf einen Prozessor, wie es bereits bei der Arbeit von Stolfo et. al [SS82] vorgeschlagen wurde. Das in [FGW84] als Condition-Level bezeichnete Verfahren zur Parallelisierung beruht auf der Verteilung der sich aus einer Production ergebenen Knoten auf mehrere Prozessoren. Entsprechend wird in [Gup86] auch der Begriff *Node-Level* Parallelisierung verwendet. Bei diesem Verfahren ergibt sich zwangsweise ein Synchronisationsaufwand zwischen den einzelnen Prozessen (bereits unabhängig von der Propagierung neu abgeleiteter Fakten), weshalb im Gegensatz zur Production-Level-Parallelisierung eine Architektur mit gemeinsam genutztem Speicher verwendet werden sollte [FGW84]. Die Action-Level Prallelisierung adressiert im Gegensatz zu den zuvor aufgeführten Verfahren nicht direkt den Match Prozess, sondern die parallele Anwendung von Änderungen an Working-Memories.

Die zuvor aufgeführte Arbeit wird von Gupta [Gup86] weiter ausgeführt und die Vor- und Nachteile der verschiedenen Ansätze werden diskutiert. So birgt insbesondere die Parallelisierung auf Ebene der Productions beispielsweise die Schwierigkeit, dass bei einer Verteilung der Production auf einzelne Prozessoren aufgrund der unterschiedlichen Belastungen keine optimale Hardwareauslastung erreicht wird. Während die Gruppierung bzw. Partitionierung von Productions dem entgegenwirken kann, stellt die Berechnung ausgewogener Partitionen eine weitere Herausforderung dar [Gup86], die z.B. von Ishida in [IS85] und [Ish90] betrachtet wird. Des Weiteren erlaubt die Production-Level-Parallelisierung aufgrund der Separierung der Berechnungen nur einen sehr geringen Grad der Optimierungen basierend auf der Netzwerkgestaltung, wie dem Node-Sharing (siehe Kapitel 4.3.1). Diese lässt sich jedoch bei der Node-Level-Parallelisierung anwen-

den, die eine (gruppierte) Verteilung der einzelnen Netzwerkknoten auf mehreren Prozessoren vorsieht.

Als weitere Parallelisierungsmöglichkeiten zeigt Gupta die *Inter-Node* Parallelisierung und die Parallelisierung der *RHS-Evaluation* auf. Die Inter-Node Parallelisierung ermöglicht eine parallele Bearbeitung von mehreren Aktivierungen eines einzelnen Knoten, um die Auswirkungen des *Kreuzprodukt-Effekts* [Gup86] zu reduzieren. Dieser beschreibt die Tatsache, dass bereits bei der Propagierung eines einzelnen Token als Eingabe für einen Beta-Knoten eine Vielzahl an Operationen durchgeführt werden müssen, um mögliche Kombination des neuen Token mit den Token des zweiten Elternknoten zu evaluieren. Die parallele RHS-Evaluation Strategie wird von Gupta als zusätzliche Möglichkeit aufgezeigt, aber nicht im Detail ausgeführt.

Die Betrachtung der zuvor aufgeführten Parallelisierungsmöglichkeiten wird von Gupta durch eine Simulation der parallelen Hardware evaluiert. Diese zeigt eine maximal zehnfache Ausführungsgeschwindigkeit gegenüber der seriellen Ausführung [Gup84] [GFNW86]. Als Grund hierfür wird einerseits die geringe Anzahl der durch eine Working-Memory Änderung betroffenen Regeln genannt und andererseits die zum Teil großen Unterschiede der notwendigen Rechenleistungen für die Verarbeitung einzelner Regeln. Um dem entgegenzuwirken wird eine möglichst feingranulare Parallelisierung empfohlen, die jedoch gleichzeitig einen höheren Kommunikations- und Synchronisationsaufwand bedingt. Unter Betrachtung dieser Aspekte wird eine Mehrprozessorarchitektur mit 32 bis 64 leistungsstarken Prozessoren vorgeschlagen, die einen gemeinsam genutzten Speicher sowie einen effizienten Hardware-Scheduler, der die Arbeitslast auf die Prozessoren verteilt, besitzt [Gup86] [GFNW86].

5.2 DATEN-PARTITIONIERUNG

Aufgrund der Tatsache, dass sich die Menge der Berechnungen für ein Production System nicht ausschließlich aus der Regelbasis ergibt, sondern sich gleichermaßen auch aus den zu verarbeitenden Daten ableitet, besteht nicht nur die Möglichkeit der parallelen Verarbeitung von Regeln, sondern auch die Möglichkeit der Partitionierung und parallelen Verarbeitung von Daten [ZWC95] [SP08b]. Um eine möglichst optimale Aufteilung der Daten zu erzielen, gilt es verschiedene Aspekte zu optimieren [SP08b]:

- Gleichmäßige Aufteilung der Arbeitslast zur Vermeidung von Wartezeiten einzelner Prozessoren, die einen Verlust von Rechenkapazität darstellt

- Kommunikationsaufwand zwischen den Partitionen, z.B. zum Austausch neu abgeleiteter Fakten

- Mehrfaches Ableiten gleicher Fakten in unterschiedlichen Partitionen
- Aufwand zur Berechnung der Partitionen selber, insbesondere bei großen Datenmengen

Die Berechnung möglichst optimaler Partitionen unter Berücksichtigung der zuvor aufgeführten Kriterien kann beispielsweise unter Verwendung von Graphen mit gewichteten Kanten, durch Hash-Funktionen, die eine einfache Zuordnung über die Berechnung eines Hash-Wertes vornehmen, oder domänenpezifisch in Abhängigkeit der jeweiligen Anwendung erfolgen [SP08b].

Insbesondere im Bereich der semantischen Datenverarbeitung werden Strategien der Daten-Partitionierung zur Anwendung von Ableitungsregeln verfolgt. Für eine möglichst optimale Hardwareauslastung wird dazu in [KOvH10] ein Ansatz vorgeschlagen, der keine feste Zuordnung von Datenfragmenten zu Prozessoren vorsieht, sondern *elastic regions* einführt. Dabei bewegen sich die Daten selbstorganisierend innerhalb von Regionen, die in Abhängigkeit der Datendichte wachsen können. In [KB13] und [PGSH14] hingegen wird eine feste Zuordnung vorgenommen, die ausgehend von einer Analyse der jeweiligen Daten erfolgt.

5.3 weitere ansätze

Bereits in Kapitel 3.3.2 wurde die Verwendung von MapReduce für die Implementierung paralleler Reasoner gezeigt. Ebenso wie für den speziellen Anwendungsfall des semantischen Reasonings werden beispielsweise in [CYZY10] und [MOK14] Ansätze vorgestellt, die basierend auf dem MapReduce-Framework eine verteilte Regelausführung implementieren. Die Ausführung der Regelverarbeitung in [CYZY10] sieht dabei einen *Master* vor, der in einem ersten Schritt eine Verteilung der Regeln (oder auch einzelne Teile einer Regel) auf verschiedene *Worker* verteilt. Die Worker leiten aus den ihnen zugeteilten Regeln ein eigenes RETE-Netzwerk ab, das in den folgenden Schritten für den Match-Prozess verwendet wird. Anschließend werden von dem Master die zu verarbeitenden Daten an die Worker propagiert, um während des Map-Schrittes den Match-Vorgang auf den jeweiligen RETE-Netzen auszuführen. Der Reduce-Schritt wird anschließend genutzt, um die Ergebnisse zusammenzuführen und die Regeln, die gefeuert werden können, zu identifizieren. Der in [CYZY10] vorgestellte Ansatz wird in [MOK14] um die zusätzliche Verwendung des Dateisystems *Hadoop Distributed File System* (HDFS) ergänzt, um eine bessere Skalierbarkeit durch eine höhere Geschwindigkeit der Datenbereitstellung zu erreichen.

Ein ähnliches Prinzip der Verteilung wird in [WZLW14] vorgeschlagen. Diese Arbeit basiert jedoch nicht auf dem Programmiermodell MapReduce, sondern nutzt ein nachrichtenbasiertes System, das ausgehend vom Austausch von Nach-

richten zwischen verschiedenen Prozessen die Bearbeitung des vollständigen RETE-Netzes gewährleistet.

Teil II

REASONING AUF MASSIV PARALLELER HARDWARE

6 PROGRAMMIERUNG PARALLELER HARDWARE

Das folgende Kapitel gibt zunächst einen Überblick über die parallele Programmierung heterogener Prozessoren unter Verwendung der Open Computing Language (OpenCL). Dabei steht nicht nur OpenCL und das damit inbegriffene Programmiermodell im Fokus, sondern auch die zugrunde liegende Architektur sowie das Speichermodell. Grundlegend dient das Kapitel dem Aufbau eines Verständnisses über parallele Architekturen und den daraus folgenden Auswirkungen auf das Ableiten paralleler Algorithmen, insbesondere für die Ausführung auf Grafikkarten. Auf dieses Wissen wird in Kapitel 7 zurückgegriffen, um den RETE-Algorithmus auf eine entsprechende Architektur und Ausführungsumgebung zu überführen. Details, die beispielsweise das Speichermodell betreffen, dienen zusätzlich als Ausgangsbasis für die in Kapitel 8 vorgestellten Optimierungen und die in Kapitel 10 vorgestellte prototypische Implementierung. Bei der Beschreibung wird zum Teil bewusst auf eine Übersetzung der Begrifflichkeiten ins Deutsche verzichtet, um den klaren Bezug zur OpenCL Spezifikation beizubehalten und die Bedeutung nicht zu verfälschen. Des Weiteren werden die OpenCL Konzepte an verschiedenen Stellen direkt anhand einer konkreten Hardware erläutert. Hierzu wurde die AMD Readon 7970 Grafikkarte gewählt, die im weiteren Verlauf der Arbeit unter anderem für die Evaluation der Forschungsergebnisse genutzt wird. Die folgenden Ausführungen beziehen sich auf die OpenCL Spezifikation 1.2, die auch als Literatur der entsprechenden Unterkapitel diente.

6.1 EINFÜHRUNG IN OPENCL

OpenCL ist ein durch die Khronos Group[1] gepflegter Standard zur einheitlichen Programmierung heterogener Parallelrechner wie Central Processing Units (CPUs), Graphics Processing Units (GPUs) und Accelerated Processing Units[2] (APUs). Die ursprünglich von Apple veröffentlichte Entwicklung wird inzwischen von vielen weiteren Unternehmen wie NVIDIA, AMD, Intel und Samsung unterstützt und liegt seit November 2013 in der Version 2.0[3] vor [Nei13]. Ein häufiges Anwen-

1 Die Khronos Group ist ein im Jahr 2000 gegründetes Industriekonsortium, das sich für eine Vielzahl offener Standards verantwortet.

2 APU bezeichnet einen Hauptprozessor mit integrierten Koprozessoren, wie z.B. einem Grafikprozessor.

3 In dieser Arbeit wird weiterhin OpenCL 1.2. verwendet, da OpenCL 2.0 zum Zeitpunkt der Niederschrift dieser Dissertation noch nicht umfassend von den Gerätetreibern unterstützt wurde.

dungsgebiet von OpenCL ist das *General Purpose Computation on Graphics Processing Units* (GPGPU), also das Ausführen von Berechnungen auf Grafikkarten.

Bei der parallelen Programmierung wird grundsätzlich zwischen *Host* und *Device* unterschieden. Der Host ist für die Koordination von Berechnungsaufgaben, den Datentransfer sowie die Synchronisation zuständig und stellt damit die Kontrolllogik zur Ausführung paralleler Berechnungen dar. Das Device bezeichnet die Hardware, auf der die tatsächlich parallele Ausführung von Code durchgeführt werden soll (z.B. GPU). In einem Ausführungskontext einer OpenCL-Anwendung steht immer genau ein Host bereit, der den Zugriff auf ein oder mehrere Devices koordiniert. Diese Trennung macht sich auch bei der Verwendung von OpenCL bemerkbar. Während für die Implementierung der Aufgaben des Hosts Programmierschnittstellen (APIs) bereitstehen, findet die Programmierung von *Kernel*-Code, also Logik, die auf der parallelen Hardware ausgeführt wird, unter Verwendung der Sprache OpenCL C statt. Die auf C99 basierende Programmiersprache beinhaltet zusätzlich einige Funktionen und Datentypen (z.B. Datentypen für Bild-Informationen) zur parallelen Verarbeitung. Gleichzeitig wurden aber auch einige Einschränkungen vorgenommen, so dass beispielsweise keine rekursiven Anweisungen möglich sind. Ein mittels OpenCL C entwickelter Kernel kann zur Laufzeit passend für die jeweils zugrunde liegende Hardware übersetzt und so für heterogene Hardware genutzt werden. Alternativ lässt sich Kernel-Code bereits im Voraus übersetzen und zu einem späteren Zeitpunkt während der Programmausführung aufrufen. In beiden Fällen ist es jedoch häufig so, dass für eine bestimmte Ziel-Plattform spezielle Kernel-Implementierungen vorgehalten werden, um die Device-abhängigen Eigenschaften optimal auszunutzen und eine bestmögliche Performance zu erzielen [MRR12]. Ein Kernel stellt selber eine „Einheit“ von Code dar, der auf einem parallelen Device ausgeführt werden kann. Entsprechend kann eine Anwendung mehrere Kernel beinhalten und diese in beliebiger Reihenfolge und Häufigkeit auf einem Device ausführen.

6.2 PLATTFORMMODELL

Das Plattformmodell beschreibt eine abstrakte Sicht auf die verwendete Hardware [MGMG11] und verbirgt damit die Heterogenität der zugrunde liegenden Architektur. Das Modell besteht aus einem CPU-basierten Host (d.h. die Host-Logik wird wie regulärer Code auf der CPU ausgeführt) und einer beliebigen Anzahl an Devices (CPUs und GPUs). Jedes Device besteht wiederum aus einer Vielzahl an Recheneinheiten (*Compute Units*), die jeweils mehrere *Processing Elements* (kurz „PE“) besitzen (vergleiche Abbildung 10). Unter Verwendung einer CPU kann eine Recheneinheit als ein *Core* bezeichnet werden, während die Recheneinheiten

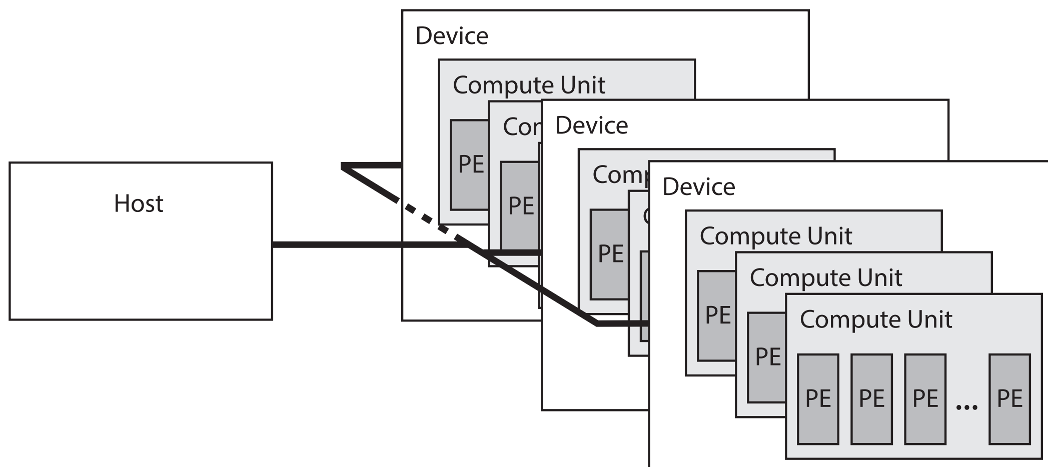


Abbildung 10: Abstrakte OpenCL Device-Architektur. Vgl. [GHK+12]

einer GPU als *Stream Cores*[4] bezeichnet werden [TS12]. Die PEs innerhalb einer Recheneinheit verarbeiten den Kernel-Code und führen somit die eigentlichen Berechnungen als *Single Instruction, Multiple Data* (SIMD) [Fly72] aus. Das heißt, dass alle Prozessoren die gleiche Operation auf unterschiedlichen Daten zur selben Zeit ausführen und so eine Parallelisierung stattfindet.

Das vorgestellte Plattformmodell entspricht dem tatsächlichen Hardware-Modell vieler Grafikkarten. Beispielsweise besteht die AMD Readon 7970 Grafikkarte aus 32 Recheneinheiten, die jeweils mit insgesamt vier 16-fachen Parallelprozessoren (insgesamt 64 PEs) bestückt sind [GHK+12]. Entsprechend kann eine Anweisung auf dem Device auf insgesamt bis zu 2048 PEs gleichzeitig ausgeführt werden (2048 Threads), wobei jede Ausführung auf unterschiedlichen Daten erfolgen kann.

6.3 Ausführungsmodell

Das Ausführungsmodell (*Execution Model*) teilt sich in zwei Bereiche auf, die bereits in der Einführung in OpenCL erwähnt wurden: *Kernel*, die auf OpenCL Devices ausgeführt werden und das *Host-Programm*, das die Ausführungsumgebung konfiguriert und bereitstellt. Das Host-Programm kann über API-Aufrufe einen *Kontext* initiieren, der als abstrakter Container die Interaktion zwischen Host und Device koordiniert und entsprechend der OpenCL Spezifikation [GM14] die folgenden vier Ressourcen verwaltet:

4 Die jeweilige Bezeichnung der Recheneinheiten kann je nach Hersteller variieren. Während AMD den Begriff der *Stream Cores* verwendet, nutzt NVIDIA beispielsweise die Bezeichnung *Stream Multiprocessor*.

- Devices: Ein oder mehrere Devices, die über die Plattform zur Verfügung stehen
- Kernel: Die OpenCL Methoden samt assoziierten Parametern, die auf einem Device ausgeführt werden
- Programmobjekte: Der Programm-Code, der die Kernel implementiert
- Speicherobjekte: Variablen, die sowohl für den Host als auch für OpenCL Devices sichtbar sind und von Kerneln zur Ausführung genutzt werden

Jedem Device innerhalb eines Kontextes ist eine *Command Queue* zugeordnet. Diese verantwortet die Kommunikation mit dem jeweiligen Device und verwaltet beispielsweise gemeinsam genutzte Speicherobjekte (bzw. den Datentransfer zu und von bestimmten Speicherregionen) sowie die Übermittlung und Ausführung von Kerneln. Zur Vorbereitung der Übergabe eines Kernels an die Command Queue wird eine *Kernel-Instanz* erstellt, die sowohl den Kernel, die Daten als Methodenparameter, die mit der Ausführung des Kernels assoziiert sind als auch Parameter, die den Indexraum (englisch: *index space*) des Kernels festlegen, beinhaltet [GM14]. Die Ausführung des Kernels erfolgt anschließend für jeden Punkt innerhalb des definierten Indexraums, wobei eine einzelne Ausführung des Kernels als ein *work-item* bezeichnet wird. Work-items einer Kernel-Instanz sind wiederum durch *work-groups* strukturiert, die eine zusätzliche Aufteilung des Indexraums vornehmen.

Der von OpenCL verwendete Indexraum nennt sich *NDRange* und besteht aus einem N-dimensionalen Raum, in dem $N = 1, 2,$ oder 3 sein kann. Der Indexraum definiert sich über die globale Größe für jede der verwendeten Dimensionen, einen Offset-Index F, der den initialen Index-Wert je Dimension vorgibt, sowie die Größe der work-groups, die ebenfalls für den N-dimensionalen Raum angegeben wird (*local size*). Basierend auf den Koordinaten innerhalb des Indexraums besitzt jedes work-item eine global eindeutige ID. Des Weiteren ist ein work-item jeweils einer work-group zugeordnet, in der die jeweilige Position innerhalb der work-group durch die *local ID* definiert wird.

Abbildung 11 verdeutlicht die beschriebenen Konzepte anhand eines Beispiels. Die Abbildung zeigt einen zweidimensionalen Indexraum, der durch die globale Größe (G_x, G_y), der work-group-Größe (S_x, S_y) und dem globalen Offset (F_x, F_y) definiert wird. Die Anzahl der work-items errechnet sich aus dem Produkt von G_x und G_y während sich die Größe der work-groups über das Produkt von S_x und S_y ergibt. Die Anzahl der work-groups wird nicht explizit angegeben, sondern errechnet sich aus der work-group-Größe sowie der Größe des globalen Indexraums. Die globale ID eines work-items ist in dem zweidimensionalen Raum durch (g_x, g_y) definiert und kann zusätzlich durch die Kombination der work-

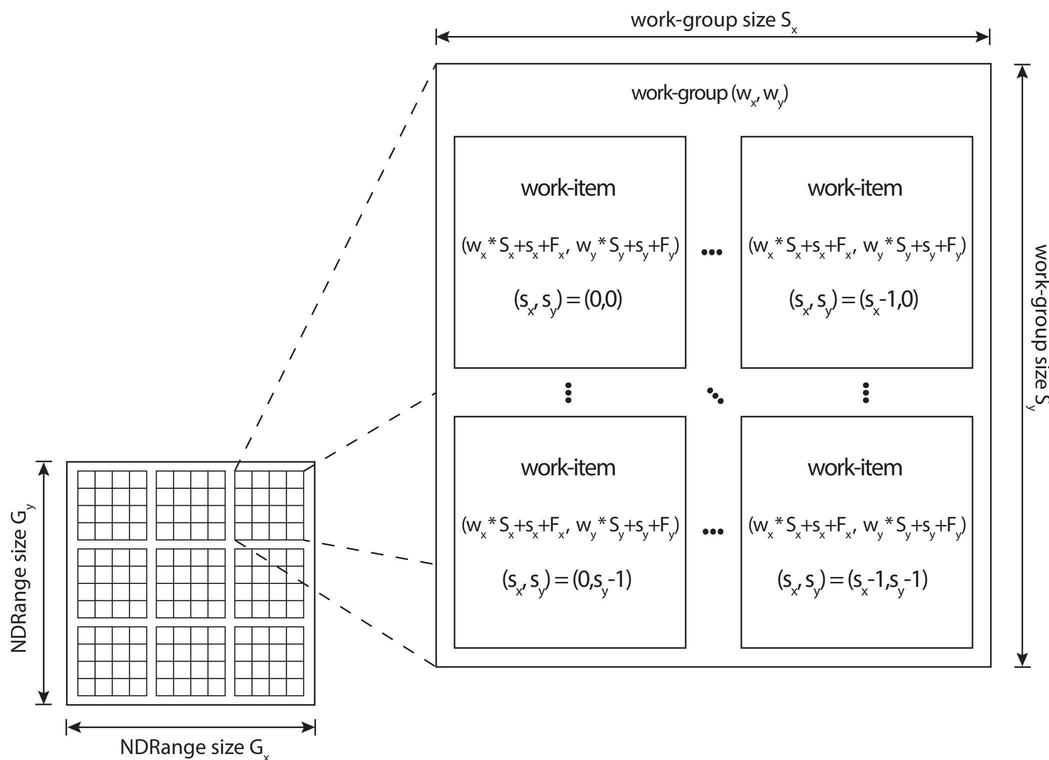


Abbildung 11: Beispiel eines zweidimensionalen OpenCL Indexraums[5]. Vgl. [GM14]

group ID (w_x, w_y), der Größe der work-group (S_x, S_y) sowie der local ID (s_x, s_y) innerhalb der work-group wie folgt errechnet werden [GM14]:

$$(g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + s_y + F_y) \tag{1}$$

Die Anzahl der work-groups lässt sich entsprechend wie folgt berechnen, wobei die Funktion $ceil(x)$ die nächste ganze Zahl liefert, die größer oder gleich dem Parameter x ist:

$$(W_x, W_y) = (ceil(G_x/S_x), ceil(G_y/S_y)) \tag{2}$$

Bei der Implementierung eines Kernel werden die zuvor vorgestellten Größen auf dem Host dazu verwendet, den Indexraum des Kernels unter Betrachtung der zu verarbeitenden Daten zu definieren. Innerhalb der Kernel-Implementierung können sie insbesondere dazu genutzt werden, die zu verarbeitenden Daten zu identifizieren (z.B. die Position innerhalb eines Daten-Arrays für ein work-item)

5 Der Abbildung liegt die Annahme zugrunde, dass sich der globale Indexraum ganzzahlig durch die work-group-Größe teilen lässt und der Offset gleich null ist.

und den Index innerhalb eines Ergebnis-Arrays zur Speicherung eines Ergebnisses zu berechnen.

6.4 SPEICHERMODELL

Aufgrund der Vielfalt der tatsächlichen Speichersysteme, die sich insbesondere bei CPUs und GPUs unterscheiden, definiert das OpenCL Speichermodell eine abstrakte Sicht, deren Mapping auf die konkrete Hardware durch den jeweiligen Hersteller verantwortet wird [GHK+12]. Grundsätzlich wird in dem Speichermodell zwischen *Global Memory*, *Constant Memory*, *Local Memory* und *Private Memory* unterschieden. Der globale Speicher ist für alle Recheneinheiten sichtbar und kann dafür genutzt werden, Daten vom Host zu einem Device zu übertragen bzw. umgekehrt auch Daten vom Device zum Host zu übermitteln (z.B. das Ergebnis einer Berechnung). Constant Memory wird als Teil des globalen Speichers modelliert und kann zur Ablage von nicht veränderbaren Objekten genutzt werden, auf die simultan aus mehreren work-items zugegriffen werden kann.

Der lokale Speicher ist jeweils ausschließlich für eine work-group sichtbar, bietet aber im Gegensatz zum Global Memory eine wesentlich kürzere Zugriffszeit und eine entsprechend höhere Bandbreite. Er wird häufig für das Zwischenspeichern von Daten genutzt, die z.B. von allen work-items einer work-group genutzt werden, um so einen langsamen Zugriff auf eine Speicherregion des globalen Speichers zu verhindern. Jedem work-item steht zusätzlich ein privater Speicherbereich zur Verfügung, in dem z.B. lokale Variablen und nicht-Pointer-Argumente abgelegt werden.

Das OpenCL Speichermodell ähnelt stark dem Speichermodell moderner Grafikkarten. Abbildung 12 zeigt auf der linken Seite das durch OpenCL definierte Speichermodell mit den zuvor erwähnten Speicherregionen. Die rechte Seite verdeutlicht, wie das Speichermodell auf eine konkrete Hardware einer AMD Readon 7970 Grafikkarte abgebildet wird. Beispielsweise wird der konstante Speicherbereich (Constant Memory) gemeinsam mit dem globalen Speicher auf den Hauptspeicher der GPU abgebildet.

Das Beispiel der verwendeten Hardware verdeutlicht zusätzlich den Unterschied der zur Verfügung stehenden Kapazität, die bei der Verwendung der unterschiedlichen Speicherregionen zu berücksichtigen ist. Entsprechend liegt eine Herausforderung bei der Entwicklung von OpenCL-Anwendungen in der Optimierung der Speicherverwendung und des Zugriffs. Beispielsweise wird bei der Verwendung von GPUs der lokale Speicher aufgrund des deutlich schnelleren Zugriffs im Gegensatz zum globalen Speicher häufig als Software-Managed Cache verwendet. Der Entwickler ist jedoch selber dafür verantwortlich, die einzelnen Speicherberei-

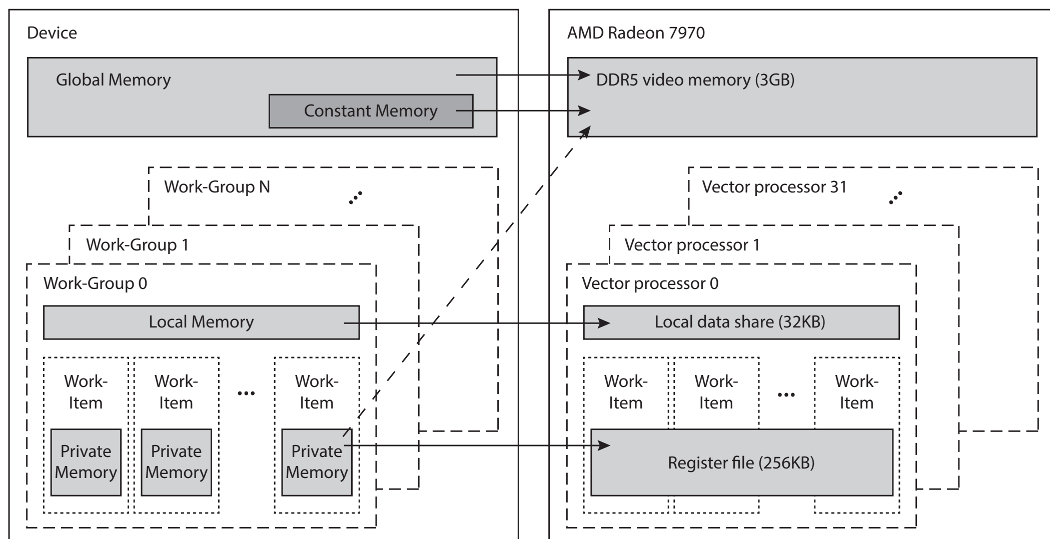


Abbildung 12: Mapping des OpenCL Speichermodells (links) auf die Architektur einer AMD Readon 7970 GPU (rechts). Vgl. [GHK+12]

che zu nutzen, Daten zwischen ihnen zu transportieren und Zugriffe zu koordinieren.

7 PARALLELISIERUNG DES RETE-ALGORITHMUS FÜR MASSIV PARALLELE HARDWARE

Für den weiteren Verlauf der Arbeit ist aufgrund der adressierten monotonen Regelunterstützung, die eine Formulierung der weit verbreiteten Ontologiesprachen und Ableitungsregeln für Large-scale Reasoning ermöglicht, die Berücksichtigung von Negationen in der Regelbasis nicht erforderlich. Entsprechend kann beispielsweise das Ableiten von neuem Wissen nicht dazu führen, dass bereits vorhandenes Wissen seine Gültigkeit verliert. Des Weiteren steht bei der adressierten Problemstellung eine besonders große Datenmenge (mehrere hundert Millionen Fakten und mehr) einer dagegen sehr kleinen Regelbasis gegenüber (z.B. 13 Regeln für die RDFS Semantik), weshalb die Verwendung eines State-saving Production-Algorithmus vor dem Hintergrund der Anzahl an möglichen Berechnungen an zusätzlicher Bedeutung gewinnt. Aus diesen Gründen wird den folgenden Überlegungen der RETE-Algorithmus als Ausgangsbasis zugrunde gelegt, der im Gegensatz zum TREAT-Algorithmus die Einträge der Working-Memories persistiert und für weitere Iterationen nutzt.

Entsprechend erfolgt in diesem Kapitel die Überführung des RETE-Algorithmus auf eine massiv parallele Ausführung. Dabei finden sowohl Konzepte der Parallelisierung von Production Systems als auch die Eigenschaften der Zielarchitektur zur Ausführung des Algorithmus Berücksichtigung. Somit wird sowohl die Forschungsfrage nach der Möglichkeit der Nutzung von massiv paralleler Hardware für den Reasoning-Prozess als auch die damit verknüpfte Verwendung eines regelbasierten Ansatzes adressiert.

7.1 BESTIMMUNG DER PARALLELITÄTSGRANULARITÄT

In dem vorherigen Kapitel wurde das OpenCL-Plattformmodell als Ausführungsumgebung für das angestrebte parallele Reasoner-Konzept unter Verwendung des RETE-Algorithmus vorgestellt. Während die tatsächliche Hardware variieren kann, vereinheitlicht die Verwendung von OpenCL den Zugriff, so dass entsprechende Anwendungen auf einer Vielzahl heterogener Hardware ausgeführt werden können. Um die Möglichkeiten einer bestimmten Zielplattform optimal auszunutzen, ist es dennoch sinnvoll, das Konzept der Parallelisierung auf eine bestimmte Hardware-Klasse auszurichten. Entsprechend zielt das folgende Konzept

vor allem auf die Ausführung auf Grafikkarten ab, die Ausführung auf anderer, von OpenCL unterstützter Hardware ist jedoch nicht ausgeschlossen.

Die Architektur moderner Grafikkarten (vgl. Abbildung 10 in Kapitel 6.2) zeichnet sich durch eine Menge an Streaming-Prozessoren (z.B. 32 für die AMD Readon 7970) und einem gemeinsam verwendbaren Speicher aus. Zusätzlich steht jedem Prozessor ein lokaler Speicher zur Verfügung, der einen besonders effizienten Zugriff gewährt. Das Scheduling der Arbeitslast wird zusätzlich durch die Hardwarearchitektur unterstützt, womit zunächst alle vier Bedingungen, die Gupta und Forgy [Gup86] [GFNW86] für die Parallelisierung des RETE-Algorithmus an die zugrunde liegende Hardware stellen, erfüllt zu sein scheinen. Aufgrund der Tatsache, dass nicht nur die einzelnen Streaming-Prozessoren einer GPU jeweils auf einer SIMD (Single Instruction Multiple Data) [Fly72] Architektur basieren, sondern auch die Prozessoren untereinander dieser Rechnerarchitektur folgen [GHK+12], ist die direkte Überführung der vorgestellten Konzepte wie Production-Level oder Node-Level-Parallelisierung auf die Ausführung auf GPUs jedoch nicht möglich. Die SIMD-Architektur bedingt, dass alle Prozessoren zu einer Zeit die gleichen Befehle (auf unterschiedlichen Daten) ausführen, was z.B. gegen eine Verteilung der RETE-Knoten auf einzelne Streaming-Prozessoren spricht, da jeder Knoten eine individuelle Verarbeitung voraussetzt. Die Argumentation der SIMD-Architektur ist auch übertragbar auf das Konzept des DADO-Ansatzes [SS82] [Sto84], der zumindest partiell eine MIMD-Verarbeitung (Multiple Instruction Multiple Data) bedingt.

Eine weitere Problematik bestünde bei einer expliziten Zuordnung der Rechenlast auf verschiedene Prozessoren weiterhin in der Lastverteilung [IS85] [Ish90], um ein möglichst optimales Laufzeitverhalten zu erzielen. Unabhängig von der Hardwarearchitektur lässt zusätzlich das OpenCL-Programmiermodell keine explizite Zuweisung von Berechnungen auf einzelne Streaming-Prozessoren zu, was ein weiterer Grund für ein alternatives Konzept der Parallelisierung ist. Die zuvor genannten Argumente sprechen ebenfalls gegen die Anwendung einer Daten-Partitionierung, der insbesondere im Hinblick auf die großen Datenmengen der hohe Aufwand für die Berechnung ausgewogener Partitionen entgegensteht.

Unter weiterer Betrachtung der Hardwarearchitektur einer GPU und dem durch OpenCL gegebenen Ausführungsmodell, das einen globalen Indexraum (G_x, G_y) beliebiger Größe[1] vorsieht, der in einen lokalen Indexraum einzelner work-groups (S_x, S_y) (z.B. (256, 1) für die AMD Readon 7970) geteilt wird, lässt sich der Grad der möglichen Parallelisierung verdeutlichen. Unter der Annahme der Verfügbarkeit von 32 Streaming-Prozessoren und einer jeweiligen maximalen work-group Größe von 256 lassen sich bereits 8192 work-items zeitgleich ausführen. Für eine

1 Die OpenCL Spezifikation trifft keine Aussage über die maximale Größe des globalen Indexraum. Eine praktische Limitierung ist abhängig von der gewählten Implementierung und den zur Verfügung stehenden Adress-Bits.

bessere Auslastung der Hardware, die beispielsweise durch Scheduling bei langwierigen Speicherzugriffen die Ausführung optimiert, kann sogar eine zum Teil deutlich höhere Anzahl an initiierten parallelen Verarbeitungen notwendig sein. Diese Zahlen verdeutlichen die Notwendigkeit einer sehr feingranularen Parallelisierung, die sich aufgrund der zuvor genannten Faktoren weder aus der klassischen Regel-Partitionierung noch aus der Daten-Partitionierung herleiten lässt. Stattdessen wird für die massiv parallele SIMD-Architektur ein Konzept benötigt, das auf einer Ebene der Ausführungslogik operiert und bei einer kleinen Regelbasis und einer gleichzeitig großen Datenmenge einen hohen Grad der Parallelisierung zulässt.

7.2 alpha-matching

Für die Durchführung des Alpha-Matchings werden zunächst aus den einzelnen Termen der gegebenen Regeln Alpha-Knoten abgeleitet. Dabei wird ein eindeutiger Regel-Term auf genau einen Alpha-Knoten abgebildet, was gleichzeitig bereits das Konzept des Node-Sharings (vgl. Kapitel 4.3.1) berücksichtigt. Das Ergebnis ist die Anzahl von Alpha-Knoten (k), der eine Vielzahl an Fakten (n) gegenüber steht. Entsprechend ergibt sich für das Alpha-Matching die Anzahl der notwendigen Match-Operationen (m) wie folgt:

$$m = k * n \tag{3}$$

Während für die Node-Level-Parallelisierung für die Alpha-Knoten k Threads erzeugt würden, wird für die Ausführung auf der GPU eine Strategie zur Erzeugung von n Threads bzw. work-items gewählt. Jedem work-item wird genau ein Datenwert zugeteilt. Für diesen iteriert das jeweilige work-item durch alle Alpha-Knoten, um die Bedingungen der Knoten mit den Eigenschaften des zugewiesenen Datenwertes zu evaluieren. Erfüllt der Datenwert die Bedingungen eines Knotens, fügt das jeweilige work-item einen Eintrag in das Working-Memory des Alpha-Knotens hinzu.

Abbildung 13 visualisiert den Indexraum, der durch m beschriebenen Berechnungen. Jedes work-item muss durch exakt k Elemente iterieren und die Überprüfung der jeweiligen Bedingungen vornehmen. Dabei ist zu beachten, dass bei den adressierten Anwendungsszenarien die Anzahl der Fakten (Tripel) die Anzahl der abgeleiteten Alpha-Knoten um ein Vielfaches übersteigt, weshalb es zu einer massiv parallelen Ausführung kommt, bei der jeder erstellte Thread eine von k abhängige Arbeitslast zu tragen hat.

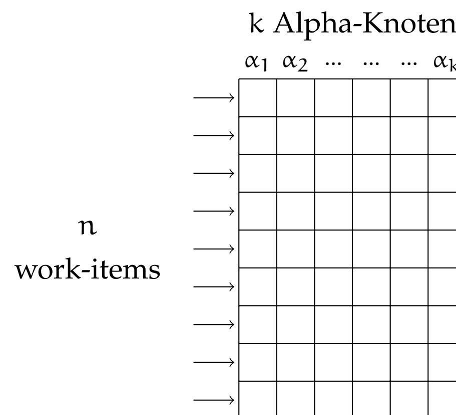


Abbildung 13: Parallelisierung des Indexraums der Berechnungen des Alpha-Matching

7.3 BETA-MATCHING

Anders als beim Alpha-Matching hängt die Anzahl und die Komplexität der notwendigen Berechnungen für das Beta-Matching nicht direkt von der Faktenbasis und dem abgeleiteten RETE-Netzwerk ab, sondern von den Working-Memories der jeweiligen Eltern-Knoten p_1 und p_2. Um eine ähnliche Parallelität wie beim Alpha-Matching zu erzielen, wird zunächst von einer Production- oder Node-Level-Parallelisierung abgesehen und lediglich die Berechnungen eines einzelnen Beta-Knotens betrachtet. Dazu sei die Menge der zu verarbeitenden Einträge eines Working-Memories x als W_x bezeichnet, während die Anzahl der Einträge als $n_x = |W_x|$ notiert werden [PBSZ13b]. Entsprechend ergibt sich für einen Beta-Knoten n_{p1} aus der Anzahl der zu verarbeitenden Einträge des Working-Memories einer der Elternknoten und n_{p2} aus der Anzahl der zu verarbeitenden Einträge im Working-Memory des zweiten Elternknoten, wobei die Annahme

$$n_{p1} \geqslant n_{p2} \tag{4}$$

gilt. Somit definiert sich die Reihenfolge der Elternknoten über die Anzahl der für eine Berechnung relevanten Einträge in den jeweiligen Working-Memories und kann während der Iterationen des RETE-Algorithmus variieren. Zur Verdeutlichung dieses Zusammenhangs führt Abbildung 14 ein Beispiel ein, das die notwendigen Berechnungen während der einzelnen RETE-Iterationen zeigt.

Während der ersten Iteration des RETE-Algorithmus ergeben sich die Eltern-Knoten p_1 und p_2 zu $p_1 = \alpha_2$ und $p_2 = \alpha_1$ direkt aus der Gesamtzahl der in den Working-Memories von α_1 und α_2 enthaltenden Einträge (Abbildung 14 a)). Werden im Rahmen der ersten Iteration neue Fakten abgeleitet, wird eine zweite Iteration des Algorithmus durchgeführt, in der beispielsweise für α_2 weitere

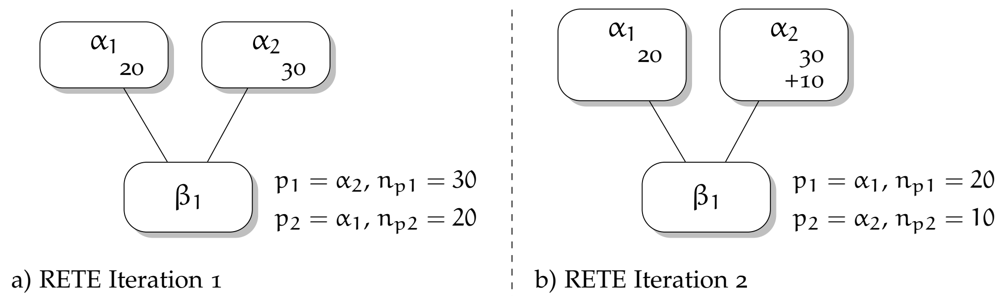


Abbildung 14: RETE-Netzwerk während zwei aufeinanderfolgenden Iterationen des RETE-Algorithmus

Matches gefunden werden. Um auch diese bei der weiteren Verarbeitung zu berücksichtigen, muss ein erneutes Beta-Matching für β_1 durchgeführt werden, das in diesem Fall ausschließlich die neuen Einträge aus α_2 und die Einträge des gesamten Working-Memories von α_1 behandelt. Entsprechend ergibt sich $p_1 = \alpha_1$ und $p_2 = \alpha_2$, um das Verhältnis $n_{p1} \geqslant n_{p2}$ sicherzustellen (Abbildung 14 b)).

Die zuvor aufgeführten Überlegungen führen dazu, dass im Falle einer Änderung an beiden Working-Memories der Eltern-Knoten ggf. zwei Schritte für die Beta-Berechnungen notwendig sind. Zusätzlich zu der in Abbildung 14 b) aufgeführten Operation müssten neue Einträge im Working-Memory von α_1 in Kombination mit allen Einträgen des Working-Memories von α_2 auf die Bedingungen von β_1 überprüft werden. Dabei findet eine erneute Zuweisung von p_1 und p_2 statt.

Der Grad der Parallelisierung für einen einzigen Beta-Knoten ergibt sich schließlich aus n_{p1}, wobei für jeden zu berücksichtigen Eintrag aus W_{p1} ein Thread erzeugt wird. Die Bedingung, dass $n_{p1} >= n_{p2}$ ist, stellt sicher, dass immer eine maximale Anzahl an Threads erzeugt wird und die Hardware der GPU optimal genutzt werden kann. Ohne die Sicherstellung dieser Bedingung können Situationen auftreten, in der sehr wenige Threads eine insgesamt hohe Arbeitslast tragen müssen, was zu einer ineffizienten Ausführung auf der GPU führen kann.

Die auf der GPU erzeugten Threads (n_{p1}) referenzieren jeweils genau ein Element aus W_{p1}, für das sie durch alle Elemente aus W_{p2} iterieren müssen. Erfüllt die Kombination der beiden Einträge aus W_{p1} und W_{p2} die Bedingungen des entsprechenden Beta-Knotens, wird ein Eintrag im Working-Memory des Knotens vorgenommen. Abbildung 15 visualisiert den Indexraum der Berechnungen. Anders als beim Alpha-Matching, muss hierbei aufgrund der möglichen kombinatorischen Explosion [AT93], auch in die X-Richtung des dargestellten Indexraums mit einer großen Anzahl an Elementen gerechnet werden (siehe auch Kapitel 4.3.1). Ähnlich wie beim Alpha-Matching wird durch die beschriebene Vorgehensweise

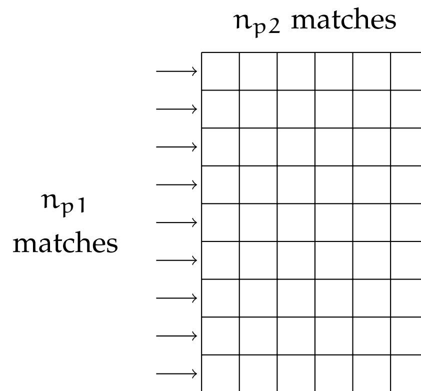


Abbildung 15: Parallelisierung des Beta-Matching für einen Beta-Knoten

ein hoher Grad der Parallelität erreicht, der in Abhängigkeit der verwendeten Regeln und der Eingabedaten eine nahezu beliebige Ausprägung annehmen kann.

Im Folgenden wird sowohl die für das Alpha-Matching als auch die für das Beta-Matching beschriebene Parallelisierung als *Device-Level-Parallelisierung* bezeichnet, womit der Ausführung auf der GPU und der Notation im Kontext von OpenCL Rechnung getragen wird.

7.4 NODE-LEVEL-PARALLELISIERUNG

Während durch die bisher beschriebenen Ansätze zur Parallelisierung lediglich die Berechnung des Alpha-Matchings bzw. die Berechnung des Matchings einzelner Beta-Knoten auf einer massiv parallelen Hardware betrachtet wurde, lässt sich eine weitere Ebene der Parallelität während des Beta-Matchings einführen. Das auf dem Prinzip der Node-Level-Parallelisierung [Gup86] basierende Konzept soll anhand eines Beispiels vorgestellt werden. Dazu werden die Regeln R3 und R4 aus der pD* Semantik [tH05] verwendet:

$$(?v \text{ owl:hasValue } ?w)\ (?v \text{ owl:onProperty } ?p)\ (?u\ ?p\ ?w) \rightarrow (?u \text{ rdf:type } ?v) \quad (R3)$$

$$(?v \text{ owl:hasValue } ?w)\ (?v \text{ owl:onProperty } ?p)\ (?u \text{ rdf:type } ?v) \rightarrow (?u\ ?p\ ?w) \quad (R4)$$

Abbildung 16 zeigt das aus den Regeln R3 und R4 abgeleitete RETE-Netzwerk, das aus insgesamt vier Alpha-Knoten und drei Beta-Knoten besteht. Wie in Kapitel 7.2 angedeutet, wurde bei der Erstellung des Netzwerks bereits das Konzept des Node-Sharings berücksichtigt. Des Weiteren zeigt Abbildung 16 die Verteilung der Beta-Knoten auf unterschiedliche Ebenen, die direkt von der Reihenfolge der Berechnungen abhängig ist. Während für die Berechnung von β_1 lediglich das Alpha-Matching abgeschlossen sein muss, hängen β_2 und β_3 zusätzlich von β_1 ab. Gleichzeitig ist zu erkennen, dass zwischen β_2 und β_3 keine Abhängigkei-

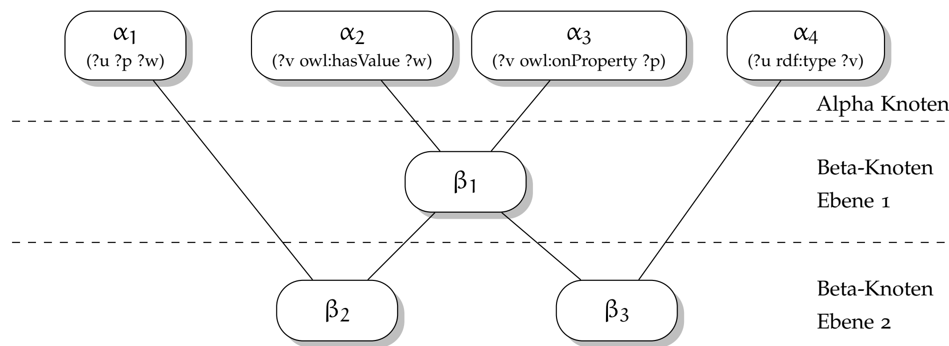


Abbildung 16: RETE-Netzwerk der Regeln R3 und R4 [PBSZ14]

ten bestehen und somit eine voneinander unabhängige Berechnung der jeweiligen Matches möglich ist. Dies gilt für alle sich auf einer Ebene befindlichen Knoten, so dass in Abhängigkeit des RETE-Netzwerks eine zusätzliche parallele Verarbeitung eingeführt werden kann, die als Ergänzung zur Device-Level-Parallelisierung dient und gleichzeitig unabhängig von der Ausführung auf einer speziellen Hardware ist. Darüber hinaus erlaubt diese Form der Parallelität auch die Verteilung der Rechenlast auf mehrere Grafikkarten, da z.B. die Berechnungen für β_2 und β_3 zeitgleich auf zwei verschiedenen GPUs ausgeführt werden können.

Analog zur Device-Level-Parallelisierung wird die zusätzlich eingeführte parallele Verarbeitung basierend auf den Beta-Knoten einer Ebene als *Host-Level-Parallelisierung* bezeichnet. Um ein möglichst hohes Maß dieser Parallelisierung zu erreichen, wird zudem die Bildung des RETE-Netzwerks als Binärbaum angestrebt, was unter Betrachtung der zuvor aufgestellten Überlegungen ergänzend zu den in Kapitel 4.3.1 aufgeführten Möglichkeiten eine Netzwerkoptimierung darstellt.

7.5 REGELAUSWERTUNG

Eine parallele Evaluation der RHS der eingesetzten Productions wurde bereits von Gupta als Möglichkeit aufgezeigt, ohne dabei eine konkrete Vorgehensweise zu beschreiben [Gup86]. Eine naheliegende Option wäre auch hier, eine der Anzahl der Productions entsprechende Menge an Threads zu erstellen, um jeweils die Ergebnisse der Terminal-Knoten zum Ableiten neuer Fakten zu nutzen. Die dabei auftretenden Probleme sind vergleichbar mit denen der Productions- bzw. Node-Level-Parallelisierung und betreffen, neben der für die Ausführung auf einer GPU ungeeigneten Granularität, vor allem die gleichmäßige Lastverteilung.

Zur Adressierung dieser Herausforderung wird das Konzept der Parallelisierung, das bereits für das Alpha- und Beta-Matching angewandt wurde, auch auf die Regelausführung übertragen. Konkret wird für jedes Element im Working-

Memory eines Terminal-Knotens ein Thread auf der GPU erzeugt. Jeder Thread ist für das Ableiten eines neuen Fakts entsprechend der RHS einer Production für einen Eintrag des Working-Memories eines Terminal-Knotens zuständig. Eine weitere Parallelisierung ist durch die zeitgleiche Verarbeitung aller Terminal-Knoten möglich, was zudem auch die Verteilung auf mehrere Devices zulässt.

Abbildung 17 zeigt das Ableiten neuer Fakten durch das beschriebene Konzept für die Terminal-Knoten der zuvor eingeführten Regeln R3 und R4. Insgesamt werden $n_{\beta 2} = |W_{\beta_2}|$ Threads bzw. work-items für das Feuern der Regel $\beta 2$, und $n_{\beta_3} = |W_{\beta_3}|$ work-items für das Feuern von $\beta 3$ erstellt.

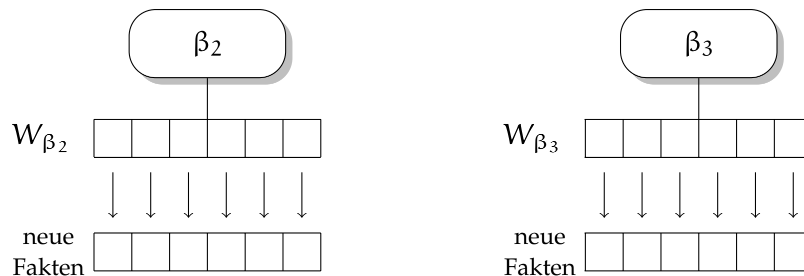


Abbildung 17: Ableiten neuer Fakten auf massiv paralleler Hardware

Das Ergebnis des beschriebenen Prozesses ist eine Menge neu abgeleiteter Fakten, die sowohl Duplikate innerhalb der eigenen Menge beinhalten kann, aber auch Duplikate in Bezug auf die bereits existierende Datenbasis. Bevor die abgeleiteten Fakten für die Weiterverarbeitung gespeichert werden, müssen beide Arten der Duplikate erkannt und entfernt werden. Heino et al. [HP12] schlagen für diesen Anwendungsfall auf der GPU die folgenden zwei Strategien vor.

Die *global duplicate removal* Strategie [HP12] sieht die Verwendung einer Index-Struktur vor, die aus den Hash-Werten für einzelne Tripel sowie dem jeweiligen Subject und Object besteht. Da der in [HP12] vorgestellte Ansatz aufgrund der Fokussierung auf die ρdf Semantik [MPG07] nur wenige Regeln implementiert, von denen ein Großteil ein `rdf:type` als Prädikat in der Ableitungsvorschrift besitzen, wird die global duplicate removal Strategie nur für diese Art der Regeln angewandt und auf die explizite Speicherung des Prädikats verzichtet. Zur Überprüfung auf Duplikate wird die Indexstruktur der gesamten Tripel in den globalen Speicher der GPU geladen, um den einzelnen work-items die Möglichkeit zu geben, die neu abgeleiteten Fakten gegenüber der existierenden Faktenbasis zu validieren.

Als weitere Möglichkeit wird in [HP12] eine *local duplicate removal* Strategie vorgeschlagen, bei der die Benennung ebenso wie bei der globalen Strategie Bezug zum relevanten Speicherbereich auf der GPU nimmt. Alle innerhalb einer work-group (vgl. Kapitel 6.3) abgeleiteten Fakten, die im lokalen Speicher abgelegt sind,

werden zunächst sortiert. Anhand der Sortierung kann jedes work-item entscheiden, ob der direkte Nachbar in der Array-Struktur ein Duplikat des eigenen Tripel ist oder nicht. Nur wenn kein Duplikat erkannt wird, wird das neu abgeleitete Tripel in den globalen Speicher kopiert und zur weiteren Verarbeitung bereitgestellt.

Beide der zuvor aufgeführten Strategien sind für die Ausführung auf der GPU ausgelegt. Die lokale Strategie erfordert jedoch zusätzlich ein nachgeschaltetes Konzept zur Vermeidung von Duplikaten, während die globale Strategie zudem die Menge der zu verarbeitenden Daten in Abhängigkeit des verfügbaren GPU-Speichers beschränkt. So kann grundsätzlich nur eine Datensatzgröße verarbeitet werden, deren Index-Struktur vollständig in dem globalen Speicher einer GPU abgelegt werden kann. Zudem müsste die Vorgehensweise für einen tatsächlich regelbasierten Ansatz dahingehend angepasst werden, dass nicht nur spezielle Regeln berücksichtigt werden (mit einem `rdf:type` als Prädikat in der RHS), sondern beliebige Duplikate erkannt werden können. Die lokale Strategie hingegen bietet das Potenzial zur Reduzierung des Aufwands während der Host-seitigen Überprüfung auf Duplikate und kann somit zu einer effizienteren Ausführung beitragen.

7.6 zusammenfassung

In den vorherigen Abschnitten wurde jeweils für die einzelnen Schritte des RETE-Algorithmus eine Möglichkeit der massiven Parallelisierung eingeführt. Die Vorgehensweise hebt sich dabei grundlegend von bisherigen Ansätzen ab, die häufig auf der Ebene des RETE-Netzwerks oder einer Datenpartitionierung operieren und versuchen, den gesamten Algorithmus bzw. den gesamten Match-Vorgang gleichzeitig auf eine parallele Architektur abzubilden. Die in dieser Arbeit vorgeschlagene Vorgehensweise sieht stattdessen eine weitere Zerlegung der Arbeitsschritte und eine darauf aufbauende, feingranulare Parallelisierung vor. So wird beispielsweise die Berechnung der Ergebnisse für einen Beta-Knoten ebenso als eine zu parallelisierende Einheit gesehen wie das Ableiten der RHS einer einzigen Regel. Die so entstehenden Blöcke der auf der GPU auszuführenden Arbeitslast müssen anschließend ggf. nacheinander auf einer GPU ausgeführt werden. Im Falle einer voneinander unabhängigen Berechnung kann auf dieser Grundlage jedoch auch eine Lastverteilung auf mehrere GPUs realisiert werden.

Die tatsächliche Parallelisierung für die Ausführung auf der GPU findet entsprechend der zuvor aufgeführten Überlegungen auf einer tieferen Anwendungs-Ebene statt, als von anderen Ansätzen vorgesehen. Im weiteren Verlauf der Arbeit wird diese Art der Parallelisierung, wie bereits eingeführt, als *Device-Level-Parallelisierung* referenziert, während beispielsweise das Konzept der Node-Level-Parallelisierung als *Host-Level-Parallelisierung* verstanden wird. Diese Notationen

tragen der jeweiligen Parallelitätsgranularität Rechnung, die in diesem Kontext auf der Host-Seite üblicherweise auf die Nutzung mehrerer Prozesse auf der CPU und auf der Device-Seite auf die Nutzung von GPUs abgebildet wird.

Während die Verwendung der Device-Level-Parallelisierung zunächst den Nachteil der (in Abhängigkeit der Anzahl der verwendeten GPUs) seriellen Ausführung der einzelnen Schritte zu besitzen scheint, wird gleichzeitig das Problem der ausgewogenen Lastverteilung, wie sie sowohl bei den Ansätzen der Regel-Partitionierung, als auch bei Ansätzen der Daten-Partitionierung auftreten, eliminiert. Eine optimale Hardwareauslastung kann nun vielmehr durch eine sorgfältige Implementierung erzielt werden, die z.B. sowohl ein effizientes Konzept für den Speicherzugriff vorsieht als auch eine für die zugrunde liegenden Hardware angepasste Größe des verwendeten globalen und lokalen Indexraums.

8 PARTITIONIERUNG UND VERTEILUNG DER ARBEITSLAST

Nachdem im vorherigen Kapitel die grundlegende Vorgehensweise für die Überführung des RETE-Algorithmus auf eine massiv parallele Ausführung beschrieben wurde, werden die eingeführten Konzepte in diesem Kapitel im Hinblick auf eine konkrete Implementierung weiter verfeinert. Dazu wird eine Möglichkeit vorgestellt, die auftretende Arbeitslast zusätzlich zu partitionieren, um eine Beschränkung der Datensatzgröße durch den verfügbaren Speicher einer GPU zu vermeiden. Gleichzeitig wird mit dem Ansatz der Partitionierung bereits die dritte Forschungsfrage dieser Arbeit, die Frage nach der Reduzierung des Ressourcenbedarfs zur Verarbeitung mehrere Milliarden Statements auf einem einzelnen Rechner, adressiert. Einen wesentlichen Beitrag zur Beantwortung dieser Frage trägt aber auch das Kapitel 9 bei, das spezielle Datenstrukturen für eine effiziente, aber gleichzeitig auch speicherschonende Ausführung des RETE-Algorithmus einführt.

Die Partitionierung der Arbeitslast erfolgt in Abhängigkeit der Eigenschaften des jeweiligen Ausführungsschritts des RETE-Algorithmus. Die sich daraus ableitende Möglichkeit der Host-Level-Parallelisierung wird anschließend in 8.3 aufgezeigt.

8.1 ALPHA-MATCHING

Das in Kapitel 7.2 eingeführte parallele Alpha-Matching basiert auf der Grundlage, dass für jedes Eingabe-Tripel genau ein work-item bzw. Thread erzeugt wird. Die Working-Memories der Alpha-Knoten werden anschließend dadurch aufgebaut, dass jeder Thread für exakt ein Tripel durch alle Alpha-Knoten iteriert und die jeweiligen Bedingungen überprüft. Unter der Annahme der Verwendung von Datensätzen mit vielen Millionen Tripel bedeutet das einerseits das Erstellen einer entsprechenden Anzahl an work-items, andererseits aber auch die Notwendigkeit der Bereitstellung aller Tripel auf der GPU.

Bei einer direkten Umsetzung dieses Konzepts ist also die Größe des Datensatzes beschränkt durch den verfügbaren GPU-Speicher, der alle Tripel aufnehmen können muss. Um diese Limitierung aufzuheben, muss eine Partitionierung des Datensatzes während des Alpha-Matchings stattfinden. Aufgrund der Tatsache, dass sich das Alpha-Matching für jedes Tripel unabhängig von anderen Tripeln ausführen lässt, kann die Gesamtarbeitslast unter Berücksichtigung der Eigenschaften der Zielhardware in kleinere Partitionen geteilt werden. Entsprechend

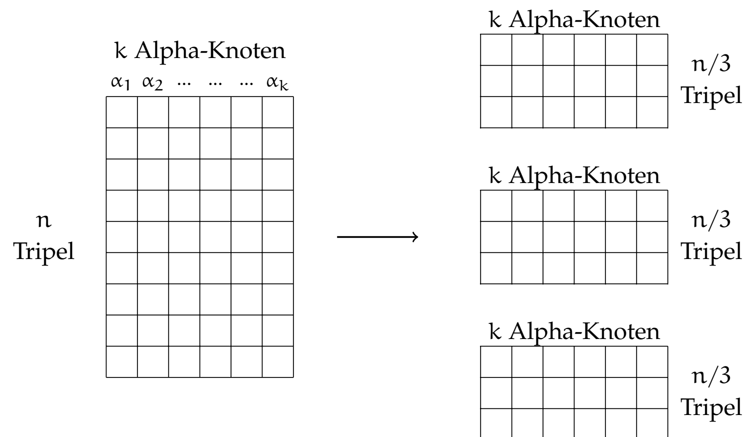


Abbildung 18: Exemplarische Partitionierung der Arbeitslast für das Alpha-Matching in drei Partitionen

der Darstellung in Abbildung 18 kann also eine Zerlegung der Eingabetripel erfolgen, bei der eine voneinander unabhängige Verarbeitung der Partitionen ermöglicht wird.

8.2 BETA-MATCHING UND REGELABLEITUNG

Eine ähnliche Vorgehensweise wie beim Alpha-Matching lässt sich grundlegend auch für das Beta-Matching und das Feuern von Regeln anwenden. Zu berücksichtigen ist hier jedoch, dass ein Eintrag eines Working-Memories üblicherweise aus Referenzen zu den tatsächlichen Fakten besteht und nicht die eigentlichen Daten enthält, was eine vielfache Redundanz der Daten hervorrufen würde. Für die Berechnung der Beta-Matches sowie für das Ableiten neuer Fakten ergibt sich daraus die Problemstellung, dass ein Working-Memory allein nicht für die Weiterverarbeitung auf der GPU ausreichend ist. Stattdessen muss zusätzlich die gesamte Faktenbasis verfügbar sein, um entsprechende Referenzen auflösen zu können.

Eine reine Partitionierung, wie sie für das Alpha-Matching vorgenommen wurde, ist aufgrund der zuvor aufgeführten Abhängigkeit nicht ausreichend, um die Beschränkung der Datensatzgröße durch den verfügbaren Speicher der verwendeten Hardware aufzuheben. Zur Adressierung dieses Problems wurde in [PBSZ14] ein Konzept unter Verwendung von *triple-matches* eingeführt, wobei ein triplematch wie folgt definiert ist:

> „*A triple-match* $m = (s, p, o, r)$ *is a quadruple with* s=*subject,* p=*predicate,* o=*object of a triple and* r=*triple reference (unique number, that is used for identification in the internal triple store).*“ [PBSZ14]

Ein triple-match besteht also nicht nur aus einem Tripel selber, sondern beinhaltet auch die intern verwendete Referenz zu diesem. Auf diese Weise können die Working-Memories weiterhin basierend auf Referenzen aufgebaut werden, die vor einer Weiterverarbeitung auf der GPU aufgelöst und in triple-matches umgewandelt werden müssen, um auf der GPU alle notwendigen Daten bereitstellen zu können.

Basierend auf den so entstehenden Datenstrukturen kann letztendlich eine Partitionierung der Arbeitslast zur Berechnung der Match-Operation für einzelne Beta-Knoten vorgenommen werden. Abbildung 20 zeigt eine exemplarische Zerlegung der Arbeitslast in der bereits bekannten Notation.

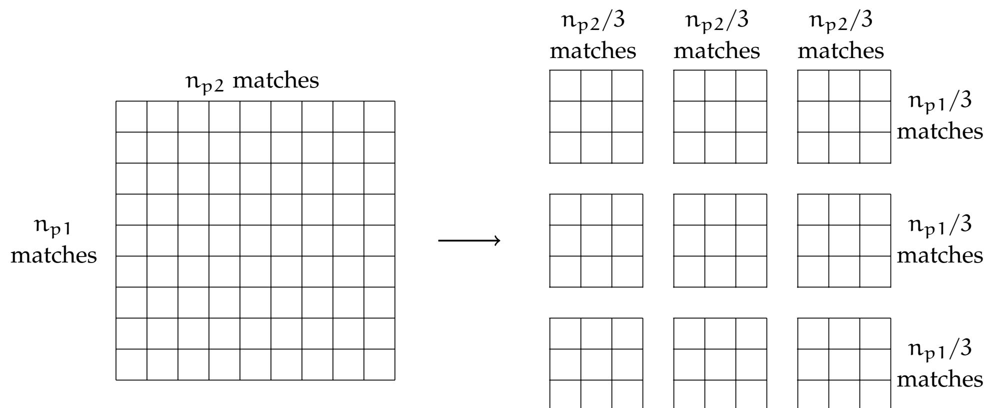


Abbildung 19: Exemplarische Partitionierung der Arbeitslast für das Beta-Matching [PBSZ14]

Der Vorgang des Ableitens von neuen Fakten, dem die Working-Memories der Terminal-Knoten zugrunde liegen, kann auf die gleiche Art partitioniert werden wie der Vorgang des Beta-Matchings. Der so abgebildete Prozess beinhaltet die Erstellung neuer Fakten und die in Kapitel 7.5 vorgestellte Duplicate-Removal Strategie. Somit werden bei der Verwendung von GPUs die innerhalb einer Work-Group erstellten Fakten zunächst sortiert, um Duplikate erkennen und bereits frühzeitig für die Weiterverarbeitung ausschließen zu können. Dieser Prozess bezieht sich jedoch jeweils nur auf einen sehr kleinen Teilausschnitt der neuen Fakten, so dass anschließend eine weitere, globale Überprüfung auf Duplikaten vollzogen werden muss, bevor die Fakten tatsächlich in die Datenbasis aufgenommen werden.

Während die Verwendung von Referenzen in Working-Memories und die darauf aufbauende Verwendung von triple-matches einerseits die Beschränkung der Datensatzgröße durch den verfügbaren Speicher einer GPU aufheben und wie in [PBSZ14] gezeigt eine performante Ausführung des RETE-Algorithmus erlauben, erfordert sie andererseits eine Speicherung aller Tripel im Hauptspeicher eines Rechners. Dies ist notwendig, um einen schnellen Zugriff während der Transfor-

mation zwischen Einträgen des Working-Memories und der triple-matches zu gewährleisten.

Unter Gesichtspunkten der ressourcenschonenden Ausführung, die entsprechend der formulierten Forschungsfragen eine Verarbeitung von bis zu mehreren Milliarden Tripeln auf einzelnen Rechnern erlauben soll, stellt die Bereitstellung aller Tripel über den Hauptspeicher eines Rechners eine große Herausforderung dar. Eine Betrachtung dieser Problemstellung wird in Kapitel 9 vorgenommen und führt dazu, dass das Konzept der triple-matches abgelöst wird, um eine weitreichende Auslagerung von Daten aus dem Hauptspeicher auf die Festplatte eines Rechners zu ermöglichen. Das in diesem Kapitel eingeführte Konzept der Partitionierung bleibt davon jedoch unberührt und kann beibehalten werden. Stattdessen ändern sich lediglich die zugrunde liegende Datenstruktur, die keine Transformation und damit auch keine Auflösung von Referenzen vor der Verwendung auf der GPU mehr erfordert.

8.3 verteilung und parallelisierung

Das zuvor eingeführte Konzept der Partitionierung erfüllt zunächst den Zweck der Zerlegung der Arbeitslast in Blöcke, die von einem Device mit beschränktem Speicher bearbeitet werden können. Gleichzeitig eröffnet es aber auch die Möglichkeit der zusätzlichen Parallelisierung und Verteilung der Arbeitslast, da die einzelnen Partitionen unabhängig voneinander bearbeitet werden können. In Kombination mit der bereits eingeführten Node-Level-Parallelisierung, die eine gleichzeitige Berechnung der Beta-Matches für Beta-Knoten einer Ebene erlaubt, kann so die Host-Level-Parallelisierung zusätzlich erweitert werden. Abbildung 20 zeigt eine mögliche Aufteilung und die dadurch resultierenden Partitionen der Arbeitslast, die sich aus dem bereits eingeführten Beispiel für β_2 und β_3 ergeben.

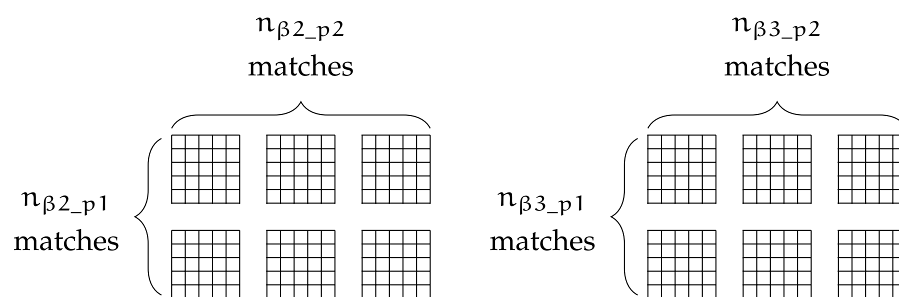


Abbildung 20: Node-Level-Parallelisierung [PBSZ14]

Die Anzahl der sich ergebenen Partitionen ist abhängig von der Menge der Beta-Knoten der einzelnen Ebenen des RETE-Netzwerks (angegeben als die Menge B), von der Anzahl der jeweiligen Einträge in den Working-Memories (n_{p1} und n_{p2}) und von der gewählten Block-Größe ($chunkheight$ und $chunkwidth$). Aus diesen Informationen lässt sich die Anzahl der möglichen parallelen Verarbeitungen bei der Annahme der Teilbarkeit von n_{p1} durch $chunkheight$ und n_{p2} durch $chunkwidth$ für eine Beta-Ebene des RETE-Netzwerks wie folgt ermitteln [PBSZ14]:

$$\sum_{i=1}^{B} \frac{n_{i_p1}}{chunkheight} * \frac{n_{i_p2}}{chunkwidth} \tag{5}$$

Während die zuvor aufgeführte Berechnung eine theoretische Betrachtung der möglichen Host-Level-Parallelität darstellt, kann in der Praxis vor allem dann ein Mehrwert aus der Zerlegung der Arbeitslast erzielt werden, wenn mehr als eine GPU in einem System verfügbar ist. In dem Fall lassen sich die einzelnen Partitionen je nach verfügbarer Rechenkapazität einer beliebigen GPU zuweisen, was im Umkehrschluss die Skalierung der Hardware ermöglicht. Dabei liegt die Annahme zugrunde, dass die verfügbaren GPUs möglichst identisch sind und sich in ihrer Leistungsfähigkeit nicht unterscheiden, da ansonsten zusätzliche Konzepte zur Berechnung einer gleichmäßigen Hardwareauslastung berücksichtigt werden müssten.

Der bereits zuvor eingeführte Begriff der *Host-Level-Parallelisierung*, der die parallele Ausführung von Berechnungen auf der CPU beschreibt, wird durch die in diesem Kapitel vorgestellte Möglichkeit der parallelen Verarbeitung der einzelnen Partitionen ergänzt. Entsprechend werden im weiteren Verlauf der Arbeit sowohl das Konzept der parallelen Verarbeitung von Beta-Knoten einer Ebene als auch die parallele Verarbeitung der erstellen Partitionen aus den Beta-Knoten zusammenfassend unter dem Begriff der *Host-Level-Parallelisierung* verstanden.

9 DATENSTRUKTUREN UND METHODEN FÜR EINE RESSOURCENSCHONENDE AUSFÜHRUNG

Die vorherigen Kapitel haben darauf abgezielt, die Arbeitslast während des Reasoning-Prozesses auf eine Weise zu zerlegen, die eine Verarbeitung von Datensätzen beliebiger Größe erlaubt. Noch keine Berücksichtigung fanden bisher die für eine Implementierung notwendigen Datenstrukturen, um Informationen wie die Tripel oder den Inhalt der Working-Memories abzubilden. Diese Informationen sind jedoch maßgeblich für den Speicherverbrauch einer RETE-basierten Reasoner-Implementierung verantwortlich und haben damit einen Einfluss darauf, welche Datensätze (im Hinblick auf die Größe) sich in Kombination mit welchem Regelsatz unter Verwendung des nur begrenzt zur Verfügung stehenden Hauptspeichers einzelner Rechner verarbeiten lassen.

Im folgenden Kapitel werden die einzelnen Schritte des Reasoning-Prozesses unter dem Aspekt der Ressourceneffizienz betrachtet und Konzepte eingeführt, um insbesondere den Speicherbedarf zu reduzieren. Somit adressiert das Kapitel vor allem den dritten Teil der Forschungsfrage dieser Arbeit, der sich mit der Ausführung des Reasoning-Prozesses für mehrere Milliarden Fakten auf einzelnen Rechnern beschäftigt.

9.1 DICTIONARY-ENCODING

Dictionary-Encoding für RDF-Daten ist ein Verfahren, bei dem die Tripel-Elemente (Subjekt, Prädikat und Objekt) durch eine numerische Repräsentation ersetzt werden. Die ersetzten Elemente werden in ein Wörterbuch (Dictionary) eingepflegt, das für jeden eindeutigen String eine eindeutige numerische Identifikation (ID) vergibt. Mehrfach vorkommende Elemente werden auf eine einzige ID abgebildet. Das Resultat eines Dictionary-Encoding-Vorgangs besteht entsprechend aus einem Dictionary sowie der numerischen Repräsentation der ursprünglichen Tripel. Zur Wiederherstellung der Tripel kann die jeweilige ID der einzelnen Tripel-Elemente verwendet werden, um mittels des Dictionaries eine Rücktransformation durchzuführen.

Das beschriebene Verfahren wird in verschiedenen Arbeiten für einen effizienten und zum Teil parallelen Reasoning-Prozess verwendet [HP12] [PBSZ14] [UKOH09] [UKM+12]. Der Grund hierfür ist einerseits, dass Dictionary-Encoding zunächst eine einfache Möglichkeit der Datenkomprimierung darstellt. Statt der

vielfachen Speicherung von Strings, die wiederkehrend in einer Datenbasis vorkommen, werden diese lediglich einmal abgespeichert und mittels einer wesentlich weniger speicherintensiven numerischen Repräsentation referenziert. Eine Evaluation eines MapReduce basierten Ansatzes für das Dictionary-Encoding, die in [UMD+13] vorgestellt wird, zeigt z.B. Kompressionsraten für unterschiedliche Datensätze, die von 7 bis 15 reichen.

Neben der Datenkomprimierung kann der Grund für die Verwendung von Dictionary-Encoding andererseits aber auch in der häufig effizienter zu verarbeitenden numerischen Repräsentation liegen. So hat diese z.B. den Vorteil, dass Vergleiche, wie beim Pattern-Matching des RETE-Algorithmus, durch einfache mathematische Operationen ausgeführt werden können und keine Operationen auf String Objekten notwendig sind [PBSZ13b]. Somit lässt sich der zugrunde liegende Rechenaufwand ebenso wie der für die Operationen notwendige Speicherbedarf reduzieren, ohne einen Verlust der Aussagekraft des durchgeführten Vergleichs einbüßen zu müssen.

Unter Berücksichtigung der Zielarchitektur (GPU), die besonders für numerische Berechnungen ausgelegt ist, wird im weiteren Verlauf der Arbeit die Verwendung von Dictionary-Encoding vorausgesetzt. Entsprechend zielen die folgenden Konzepte sowie die anschließende Implementierung der einzelnen RETE-Operationen auf vorverarbeitete und Dictionary-kodierte Datensätze ab.

9.1.1 *Benötigte Daten und Aufbau einer naiven Implementierung*

Mit der Zielsetzung, große Datensätze auf einer möglichst einfachen Hardware verarbeiten zu können, geht auch das Parsen und Kodieren der Daten als Schritt der Vorverarbeitung auf selbiger Hardware einher. Bereits bei diesem Prozess kann es aufgrund beschränkter Ressourcen zu einer Begrenzung der verarbeitbaren Datensatzgröße kommen. Im Folgenden wird deshalb zunächst eine theoretische Betrachtung des Dictionary-Encodings samt der dazu notwendigen Datenstrukturen durchgeführt, bevor im nächsten Unterkapitel eine Methode zur Reduzierung des Speicherverbrauchs vorgestellt wird.

Eine minimale Implementierung des Dictionary-Encodings muss einerseits das Kodieren von String-Informationen zu einer numerischen Repräsentation unterstützen und andererseits eine Möglichkeit der Rückwandlung bieten. Eine mögliche Struktur für die Realisierung dieser Funktionalität ist in Abbildung 21 dargestellt. Die Datenstrukturen bestehen aus einem HashSet und einem Dictionary, das beispielsweise als ArrayList aufgebaut werden kann und die eigentlichen String-Informationen enthält. Das HashSet dient der Identifizierung bereits vorhandener Strings. Dazu wird beim Hinzufügen eines neuen Elements der jeweilige Hash-Code berechnet und basierend auf dem HashSet ein Lookup durchgeführt, ob für

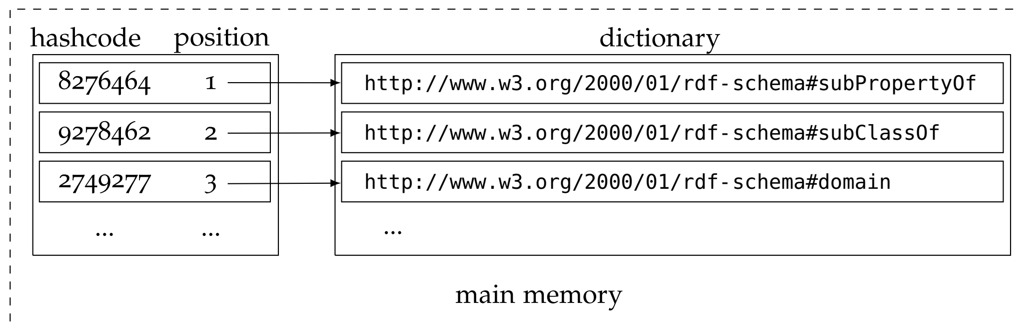


Abbildung 21: Datenstrukturen für das Dictionary-Encoding

den Hash-Code bereits ein Eintrag existiert. Ist die identifizierte Spalte in dem HashSet noch frei, wird ein neuer Eintrag erstellt, dessen Wert auf die Position des Dictionaries zeigt, an der der String eingefügt wird. Ist an der identifizierten Stelle im HashSet jedoch bereits ein Wert vorhanden, kann es sich entweder um einen identischen String oder um eine Hash-Kollision handeln. In beiden Fällen muss der durch die belegte Zeile referenzierte String betrachtet werden, um entweder die Übereinstimmung festzustellen und die numerische Repräsentation zurückzugeben oder eine Kollisionsbehandlung durchzuführen.

9.1.2 *Reduzierung der Hauptspeicherbelastung*

Das zuvor eingeführte Konzept des Dictionary-Encoding sieht die vollständige Speicherung der notwendigen Datenstrukturen im Hauptspeicher eines Rechners vor. Besonders speicherintensiv ist dabei das Dictionary selber, das alle String-Daten beinhaltet, die je nach Datensatz aus unterschiedlich großen Einträgen bestehen können. Neben einfachen RDFS-Attributen, wie sie in Abbildung 21 dargestellt sind, können auch Texte mit einer nahezu beliebigen Länge gespeichert werden. Somit wird die Anzahl der verarbeitbaren Tripels während des Parsens maßgeblich von dem verfügbaren Speicher und dem Dictionary bestimmt.

Um die Belastung des Hauptspeichers zu reduzieren, kann eine Auslagerung der String-Daten auf die Festplatte vorgenommen werden [PSZ15]. Abbildung 22 stellt dazu eine modifizierte Form der bereits bekannten Datenstrukturen vor, die zusätzlich um Strukturelemente zur effizienten Identifizierung einzelner Dictionary-Einträge auf der Festplatte ergänzt wurden. Statt der Speicherung der String-Daten wird lediglich eine Präfix-Summe über die jeweilige Länge der einzelnen Einträge erstellt und im Hauptspeicher vorgehalten. Die String-Einträge können aufeinanderfolgend in eine Datei geschrieben werden. Um einen String an der Stelle i aus der Datei zu lesen, kann der Dateizeiger an die Stelle $k(i-1)$ ($k(i)$ definiert den Wert an der Stelle i in der Präfix-Summe) navigiert werden, um

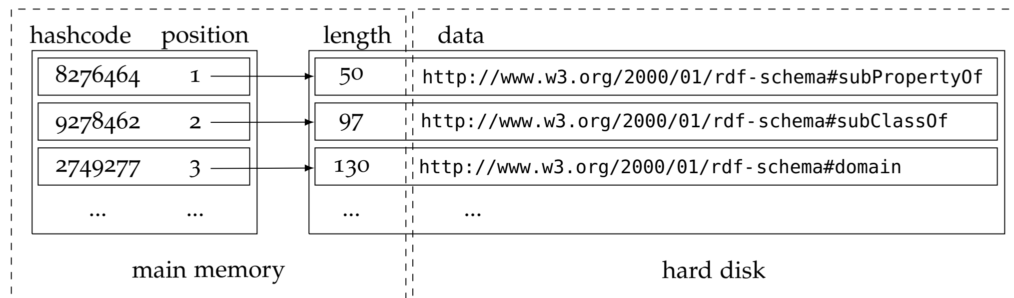


Abbildung 22: Datenverteilung beim Dictionary-Encoding unter Verwendung des Festplattenspeichers

anschließend $k(i) - k(i-1)$ Zeichen zu lesen. Durch die Verwendung der Präfix-Summe wird eine effiziente Lokalisierung des jeweiligen Strings ermöglicht, so dass eine Anfrage an das Dictionary mit lediglich einer lesenden Operation auf der Festplatte beantwortet werden kann.

Der Aufbau dieser Strukturen birgt jedoch noch weitere Vorteile. Nach Abschluss des Parse-Vorgangs der Eingabedaten werden die aufgebauten Strukturen ausschließlich für die Decodierung benötigt. Die Zuordnungen der Hash-Codes zu den jeweiligen Positionen im Dictionary werden somit obsolet und können aus dem Speicher entfernt werden. Weiterhin kann auch die Präfix-Summe in eine Datei geschrieben werden, um sämtlichen Speicher, der für die Vorverarbeitung notwendig ist, für den folgenden Reasoning-Prozess freizugeben. Aufgrund der typischerweise konstanten Byte-Länge der verwendeten numerischen Repräsentation der jeweiligen Summen-Elemente (z.B. 8 Byte) kann eine Decodierung im Anschluss durch eine zusätzliche Lese-Operation auf der Festplatte ausgeführt werden. Das dabei anzuwendende Verfahren ist identisch zu der beschriebenen Lese-Operation für Strings. Die Position für den Dateizeiger ergibt sich direkt aus der verwendeten Anzahl an Bytes je Präfix-Summen-Eintrag sowie aus der Position i des angeforderten Strings zu $(i-1) * 8$ (für eine 8 Byte Repräsentation der Summenelemente). An der berechneten Stelle müssen exakt zwei Zahlenwerte gelesen werden, um die Differenz der Summen zur Ermittlung der String-Länge sowie die Position des Strings auf dem Dateisystem berechnen zu können.

Das vorgestellte Verfahren zum Dictionary-Encoding ermöglicht die Reduzierung des Hauptspeicherbedarfs während des Parsens und die vollständige Freigabe von benötigtem Speicher nach der Verarbeitung aller Eingabedaten. Die aus dem Konzept resultierenden Zugriffe auf die Festplatte sind jedoch zeitaufwändiger als der Zugriff auf Daten des Hauptspeichers. Um den ressourcenintensiven Festplattenzugriffen entgegenzuwirken kann zusätzlich ein Caching-Mechanismus implementiert werden, der zur Beantwortung besonders häufig auftretender Codierungs- oder Decodierungsanfragen verwendet wird. Ein tendenziell hohes Auf-

kommen an wiederkehrenden Anfragen kann beispielsweise für die Schema-Informationen (TBox) eines Datensatzes erwartet werden, die die Struktur der Daten definieren.

9.2 WORKING-MEMORIES

Die Verwendung des Festplattenspeichers zur Auslagerung von Daten wurde bereits in den vorherigen Kapiteln als eine Möglichkeit zur Reduzierung des Ressourcenbedarfs eingeführt. Während dieser Speicher in der Regel ein wesentlich größeres Fassungsvermögen aufweist, ist der Zugriff darauf deutlich langsamer. Entsprechend sollten bei der Auslagerung von Informationen auf die Festplatte die verwendeten Datenstrukturen so gestaltet sein, dass notwendige Daten zusammenhängend und mit einer möglichst geringen Anzahl an Operationen gelesen und geschrieben werden können.

Das in [PBSZ14] verwendete Konzept der triple-matches ermöglicht bereits die Auslagerung der Working-Memories auf die Festplatte. Anschließende Lese-Operationen können in Blöcken durchgeführt werden, um beispielsweise alle in einem Working-Memory enthaltenen Referenzen zu lesen. Die darauf folgende Transformation von n Einträgen eines Working-Memories zu triple-matches erfordert während der Auflösung der Referenzen einen Zugriff auf mindestens n nicht aufeinander folgende Tripel (bei Beta-Knoten kann sich die Anzahl auf ein Vielfaches von n erhöhen), weshalb bei der Verwendung dieses Konzepts die Tripel im Hauptspeicher gehalten werden müssen. Eine zusätzliche Auslagerung der Tripel auf die Festplatte würde zu massiven Festplattenzugriffen führen und eine Ausführung extrem verlangsamen.

Um einen effizienten Zugriff zur Bereitstellung der notwendigen Informationen aus den Working-Memories für die verschiedenen Schritte des RETE-Algorithmus gleichermaßen zu gewährleisten, wie die Möglichkeit der Auslagerung der Daten auf die Festplatte, wird im Folgenden ein Konzept zur Gestaltung der Working-Memories ohne die Verwendung von Referenzen eingeführt.

Statt dem Aufbau der Working-Memories basierend auf Referenzen, werden die eigentlichen Tripel-Daten verwendet und eine Replikation der Informationen akzeptiert. Abbildung 23 verdeutlicht den Unterschied beider Ansätze anhand der bereits in Kapitel 4.2 eingeführten Regeln R_1 und R_2, die für eine bessere Übersicht im Folgenden noch einmal aufgeführt sind:

$$(?x \;\; ?p \;\; ?y) \rightarrow (?p \;\; \text{rdf:type} \;\; \text{rdf:Property}) \tag{R_1}$$

$$(?x \;\; ?p \;\; ?y), \;\; (?p \;\; \text{rdfs:domain} \;\; ?c) \rightarrow (?x \;\; \text{rdf:type} \;\; ?c) \tag{R_2}$$

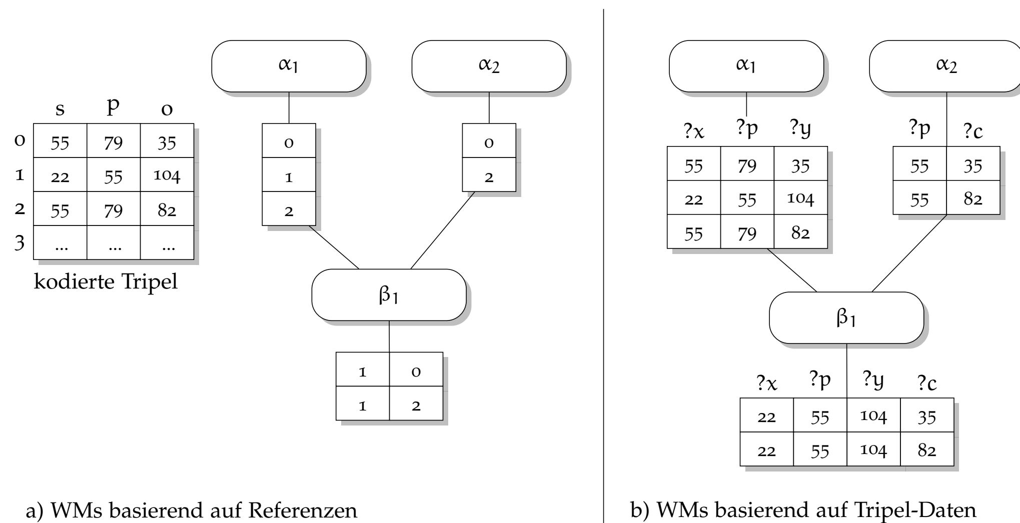


Abbildung 23: Unterschiedliche Aufbau-Möglichkeiten der Working-Memories für die RETE-Knoten der Regeln R_1 und R_2. Vgl. [PSZ15]

Während bei der Verwendung von Referenzen die Einträge der Working-Memories aus lediglich wenigen Datenelementen bestehen (abhängig von der jeweiligen Ebene eines Knotens im RETE-Netzwerk), müssen zusätzlich die Tripel in kodierter Form zur Auflösung der Referenzen bereitgestellt werden (vgl. Abbildung 23 a)). Der Aufbau der Working-Memories basierend auf den Tripel-Daten hingegen resultiert in Einträgen, die sich häufig aus mehreren Datenelementen zusammensetzen. Für α_1, dessen Regel-Term aus drei Variablen besteht, müssen beispielsweise alle drei Tripel-Elemente im Working-Memory aufgenommen werden. Der Knoten α_2 hingegen referenziert einen Regel-Term mit zwei Variablen. Entsprechend besteht ein Eintrag des Working-Memories auch aus zwei Datenelementen. Bei den Beta-Knoten werden ebenfalls nur Daten im Working-Memory gespeichert, die nicht statisch sind (es werden nur Werte von Variablen gespeichert) und keine redundanten Informationen darstellen (beispielsweise das mehrfache Speichern des Wertes ?p für β_1). Entsprechend ist die Anzahl der Datenelemente pro Eintrag in einem Working-Memory direkt abhängig von den jeweiligen Knoten, die auf die Regel-Terme zurückzuführen sind.

Das vorgestellte Konzept führt aufgrund der durchschnittlich höheren Anzahl an Datenelementen pro Working-Memory-Eintrag zu einem insgesamt gesteigertem Speicherverbrauch. Der notwendige Speicherplatz kann aufgrund der Struktur der Daten jedoch auch von einer Festplatte gedeckt werden, da der Aufbau der Daten das Lesen und Weiterverarbeiten in Blöcken erlaubt. Auf diese Weise können z.B. Informationen, die für die Ausführung einer Partition der Arbeitslast (vgl. Kapitel 8) erforderlich sind, mit einer einzigen Lese-Operation geladen

werden. Des Weiteren müssen die Tripel selber nicht mehr im Speicher gehalten werden, um eine effiziente Auflösung von Referenzen zu gewährleisten. Somit lässt sich der Speicherverbrauch für die Verwaltung von Working-Memories vollständig auf die Festplatte auslagern und die Belastung des Hauptspeichers weiter reduzieren.

9.3 TRIPEL-INDEX-STRUKTUR ZUR ERKENNUNG VON DUPLIKATEN

In den vorherigen Kapiteln wurde gezeigt, wie die Belastung des Hauptspeichers während des Dictionary-Encodings sowie während des Reasoning-Prozesses durch die Auslagerung der Working-Memories reduziert werden kann. Der Reasoning-Prozess bedingt jedoch nicht nur die Speicherung der Working-Memories, sondern benötigt weitere Datenstrukturen zur Erkennung von mehrfach abgeleiteten Tripeln (Duplikaten). In [PSZ15] wurde beispielsweise gezeigt, dass in Abhängigkeit der verwendeten Regeln sowie des verwendeten Datensatzes 98,8% der abgeleiteten Tripel Duplikate sein können, was zu einer Materialisierung von rund 580 Millionen Tripel geführt hat, von denen 573 Millionen wieder verworfen wurden. Diese Zahlen verdeutlichen die Notwendigkeit einer effizienten Deduplikations-Strategie in Bezug auf die Ausführungsgeschwindigkeit, die eine Verwendung des Hauptspeichers bedingt.

9.3.1 *Aufbau einer naiven Implementierung*

Eine einfache Implementierung zur Erkennung von Duplikaten kann ebenso wie bereits das Dictionary-Encoding auf einer Hash-Struktur basieren. Beispielsweise kann ein HashSet entweder die Speicherung der Tripel selber oder die Speicherung der Position eines Tripels als Wert vorsehen. Die zuletzt genannte Vorgehensweise macht insbesondere dann Sinn, wenn die Tripel ohnehin im Hauptspeicher gehalten werden müssen. Zur Identifizierung von Duplikaten muss für jedes abgeleitete Tripel der Hashwert berechnet werden, um die entsprechende Position mit einer möglichen Referenz auf ein bereits bestehendes Tripel zu ermitteln. Sofern der Eintrag im HashSet belegt ist, muss das referenzierte Tripel geladen werden, um zu überprüfen, ob es sich um ein Duplikat oder eine Hash-Kollision handelt. Im Falle einer Kollision erfolgt eine entsprechende Behandlung.

Mit den vorgestellten Datenstrukturen kann eine Überprüfung auf die Existenz eines Tripels unter der Annahme, dass keine Kollisionen auftreten, mit lediglich einer Abfrage innerhalb des HashSets erfolgen, die ggf. zusätzlich das Laden eines Tripels aus einer Array-Struktur bedingt. Somit kann bereits aufgrund des überschaubaren Rechenaufwands mit einer konstanten Ausführungsgeschwindigkeit für die Identifizierung eines einzelnen Tripels kalkuliert werden. Anderer-

seits bedingt die vorgestellte Struktur mindestens die Speicherung aller Tripel im Hauptspeicher, was aufgrund der zuvor eingeführten Strukturen für die Working-Memories nicht mehr erforderlich ist und somit ausschließlich dem Zwecke der Deduplikation dient. Die resultierende (minimale) Speicherbelastung lässt sich bei der direkten Verwendung der Tripel zum Aufbau des HashSets entsprechend in Abhängigkeit der Anzahl der verfügbaren Tripel n sowie dem für das HashSet gewähltem Load-Faktor f berechnen. Bei einer maximalen Auslastung des HashSets ergibt sich unter Verwendung von 8 Byte Datentypen für die numerische Repräsentation der Tripel eine Gesamtbelastung von $(\frac{n}{f}) * 3 * 8$ Byte. Bei einem zugrunde liegendem Load-Faktor von $0,7$ werden somit für die Speicherung eines einzelnen Tripels für die Deduplikation mindestens 34,3 Byte benötigt. Hinzu kommen weitere Bytes, sofern das HashSet nicht vollständig gefüllt ist, wodurch bei einer effektiven Belastung von 60% mit rund 40 Byte pro Tripel gerechnet werden muss.

9.3.2 *Komprimierte Speicherung von Tripeln*

Während der Informationsgehalt der aufgeführten Daten zwingend für eine Deduplikation erforderlich ist, besteht dennoch die Möglichkeit der Komprimierung. Diese muss jedoch die Bedingung erfüllen, dass anders als bei vielen Komprimierungsverfahren für numerische Werte, keine Sequenzen von Zahlen zur Komprimierung verwendet werden. Diese würden bei einem einfachen Zugriff eine Dekomprimierung der gesamten Sequenz bedingen, was einen ungeordneten Zugriff auf einzelne Werte verlangsamen würde.

Eine einfache Möglichkeit der Komprimierung für den speziellen Fall der Tripel ist das *Vertical-Partitioning* [AMMH07]. Vertical-Partitioning basiert auf der Erkenntnis, dass viele Datensätze trotz einer großen Anzahl an Fakten häufig durch nur wenige Prädikate beschrieben werden [ÁGBFMP11] [PSZ15]. In [ÁGBFMP11] wurde beispielsweise anhand einer Evaluation gezeigt, dass die dort verwendeten Datensätze aus verschiedenen Domänen trotz einer Größe von 9 Millionen bis über 232 Millionen Fakten häufig von nur wenigen hundert Prädikaten beschrieben sind. Der größte und gleichzeitig heterogenste Datensatz ist DBpedia, dessen 232,5 Millionen Tripel knapp 40.000 unterschiedliche Prädikate besitzen. Um diese Beobachtung für die Datenkomprimierung zu nutzen, wird beim Konzept des Vertical-Partitioning bei der Datenrepräsentation der Tripel auf die explizite Speicherung der Prädikate verzichtet. Stattdessen werden die Tripel z.B. in mehrere Listen aufgeteilt, die jeweils einem Prädikat zugeordnet sind. Auf diese Weise kann bei einem Datensatz wie UniProt, der trotz der Größe von über 72 Millionen Fakten durch nur 126 Prädikate beschrieben wird [ÁGBFMP11], unter Vernachläs-

sigung des ggf. notwendigen zusätzlichen Speichers für die Erstellung mehrerer Listen rund 33% des Speicherbedarfs eingespart werden.

Unter Berücksichtigung des Vertical-Partitioning bleiben pro Tripel noch zwei (numerische) Werte übrig, die komprimiert werden können. Hierzu bieten sich klassische Verfahren aus dem Bereich der 32 bzw. 64 Bit Integer-Komprimierung an[1], über die die Arbeit von Lemire und Boytsov [LB13] einen guten Überblick gibt. Für den konkreten Fall der in dieser Arbeit als *Tripel-Index-Struktur* bezeichneten Datenstruktur wird eine Kombination aus den beiden Verfahren der *Differenzkodierung* und der *Variablen-Byte-Kodierung* [WZ99] gewählt. Die Differenzkodierung bietet sich insbesondere für sortierte Sequenzen von numerischen Werten an und sieht vor, statt der eigentlichen Zahl lediglich die Differenz zur vorherigen Zahl zu speichern. Statt der Speicherung des Subjekts und Objekts (s, o), kann unter der Bedingung, dass $s > o$ und $((s - o) < o)$ z.B. $(s, s - o)$ gespeichert werden. Ein Anwenden der Differenzkodierung ist entsprechend der aufgeführten Bedingung nur dann sinnvoll, wenn der berechnete Wert auch tatsächlich kleiner als der Ausgangswert und das Ergebnis nicht negativ ist. In dem Fall ermöglicht es jedoch in den folgenden Schritten eine effizientere Speicherung der Daten. Die resultierenden Zahlen werden anschließend unabhängig davon, ob die Differenzkodierung angewandt wurde oder nicht, einem rechten Wert v_r und einem linken Wert v_l zugeordnet. Dabei wird sichergestellt, dass der rechte Wert immer kleiner oder gleich dem linken ist. Die Vorgehensweise zur Berechnung von v_l und v_r ist in Algorithmus 1 beschrieben.

Die Anwendung des bisher beschriebenen Konzepts bewirkt ggf. eine Verkleinerung des Zahlenwertes sowie eine Sortierung der Werte entsprechend ihrer Größe. Diese Maßnahmen haben bei einem zugrunde liegenden 64 Bit Datentyp jedoch noch keine Auswirkung auf den notwendigen Speicher und stellen somit auch keine Komprimierung dar. Stattdessen muss zusätzlich ein Konzept etabliert werden, das eine Speicherung von numerischen Werten mit exakt der Anzahl an Bytes erlaubt, die auch tatsächlich zur Repräsentation des Zahlenwerts benötigt werden. Zu diesem Zweck wird das Verfahren der Variablen-Byte-Kodierung adaptiert und angewandt, das eine Speicherung von Zahlensequenzen mit jeweils einer minimalen Anzahl an Bytes erlaubt. Dazu werden die ersten sieben Bit eines Bytes genutzt, um die eigentlichen Daten zu repräsentierten. Das achte Bit dient als Indikator, ob das folgende Byte ebenfalls noch zu einem Datenwert gehört oder bereits einen neuen Wert im genutzten Byte-Array darstellt [LB13]. Konkret definiert eine 1 als achtes Bit das Ende eines Wertes, während eine 0 die Fortführung anzeigt.

Die Werte im Intervall $[0, 2^7)$ können beispielsweise durch ein einziges Byte kodiert werden, bei dem die ersten sieben Bit zur Speicherung des jeweiligen Wertes

1 Aufgrund der Ausrichtung dieser Arbeit auf Large-scale-Datensätze wird im Folgenden eine 64 Bit Repräsentation der Zahlenwerte zugrunde gelegt.

Algorithmus 1 : Berechnung der Werte v_l und v_r [PSZ15]

```
Data : subject: s, object: o
Result : value left: v_l, value right: v_r
v_l = s;
v_r = o;
if s > o then
    if (s - o) < o then
        v_r = s - o;
    end
else
    if (o - s) < s then
        v_r = o - s;
        v_l = o;
    else
        v_l = o;
        v_r = s;
    end
end
```

dienen und das achte Bit mit einer 1 das Ende des Wertes anzeigt. Entsprechend können die Zahlen $[2^7, 2^{14})$ mittels zwei Byte kodiert werden, bei denen das achte Bit des ersten Byte eine 0 ist und das achte Bit des zweiten Byte eine 1. Zur tatsächlichen Speicherung des Datenwerts stehen die übrigen 14 Bit zur Verfügung [LB13] [PSZ15]. Eine Kodierung des Wertes 42 resultiert entsprechend dieser Vorgehensweise in der Binärdarstellung

$$1\,\underbrace{010\ 1010}_{\text{Daten}}$$

während der Wert 4200 als

$$1\,\underbrace{0010\ 0000}_{\text{Daten}}\ 0\,\underbrace{110\ 1000}_{\text{Daten}}$$

repräsentiert wird. Durch die 1 als Indikator für das Ende eines Datenwerts an der Stelle des Bits mit der höchsten Wertigkeit, können durch die so vorgenommene Kodierung mehrere Werte hintereinander in einem Byte-Array gespeichert werden, wobei jeder Zahlenwert nur die tatsächlich benötige Anzahl an Bytes nutzt.

Um dieses Konzept auf den Anwendungsfall der Tripel-Index-Struktur zu übertragen, bei der eine zu komprimierende Zahlensequenz aus den beiden Werten v_r und v_l besteht, ist es ausreichend, lediglich den Wert v_r entsprechend der beschriebenen Vorgehensweise zu kodieren. Alle nach einer ersten 1 an der achten Stelle eines Bytes folgenden Bits können direkt dem Wert v_l zugeordnet werden. Aus die-

sem Grund findet auch eine Zuordnung der zu kodierenden Werte entsprechend ihrer Größe zu v_l und v_r statt (siehe Algorithmus 1), um sicherzustellen, dass die speicherintensivere Kodierung auf den kleineren Zahlenwert angewandt wird. Zusätzlich muss jedoch die Information hinterlegt werden, ob eine Differenzkodierung angewandt wurde und ob die Reihenfolge der kodierten Werte dem Subjekt als linker Wert und dem Objekt als rechter Wert entspricht. Zu diesem Zweck werden die beiden höchstwertigsten Bits des Byte-Arrays als Flag genutzt.

Die beschriebene Vorgehensweise soll anhand eines Beispiels mit den Werten $s = 622$ und $o = 35$ verdeutlicht werden. Aufgrund der Tatsache, dass die Differenz zwischen s und o größer ist als o selber ($(s - o) > o$), wird keine Differenzkodierung vorgenommen. Somit ergeben sich $v_l = s = 622$ und $v_r = o = 35$. Die Binär-Repräsentation von 35 ist 0010 0011, was unter Anwendung der Variablen-Byte-Kodierung zu 1010 0011 führt. Die Binär-Repräsentation von 622 entspricht 0000 0010 0110 1110, wobei die beiden höchstwertigsten Bits die Information beinhalten, ob Differenzkodierung angewandt wurde und in welcher Reihenfolge das Subjekt und Objekt gespeichert sind. Da weder Differenzkodierung angewandt wurde (höchstwertigstes Bit) noch ein Tausch der Werte v_l und v_r stattgefunden hat (Bit mit der zweithöchsten Wertigkeit), wird beiden Bits eine 0 zugeordnet. Durch das Zusammenfügen der kodierten Informationen ergibt sich die folgende Bitfolge: [PSZ15]

$$00\underbrace{00\ 0010\ 0110\ 1110}_{v_l}\ 1\underbrace{010\ 0011}_{v_r}$$

Die komprimierten Informationen aus dem Beispiel lassen sich durch Anwendung der vorgestellten Konzepte mit lediglich drei Byte speichern. Dem gegenüber stehen 16 Byte, die für eine Speicherung ohne Komprimierung (aber unter Berücksichtigung des Vertical-Partitioning) notwendig wären. Somit werden für das aufgeführte Beispiel weniger als 20% des ursprünglichen Speichers benötigt, ohne dabei den Informationsgehalt der Daten zu reduzieren. Die Kompressionsrate ist jedoch abhängig von den konkreten Zahlenwerten, die ein Tripel bilden und kann entsprechend variieren.

9.3.3 *Zusammenfasung*

Das vorgestellte Konzept zur komprimierten Speicherung von Tripel-Informationen für den Vorgang der Deduplikation basiert auf den Verfahren des Vertical-Partitioning, der Differenzkodierung und der Variablen-Byte-Kodierung. Durch die Kombination und Adaption dieser Methoden kann der Speicherverbrauch für die Repräsentation eines einzelnen Tripels, das bei einer Dictionary-kodierten Darstellung aus drei Zahlenwerten und insgesamt 24 Byte besteht, um bis zu 90% ge-

senkt werden. Aufbauend auf diesen Konzepten kann das eingangs beschriebene Verfahren zur Erkennung von Duplikaten basierend auf einem HashSet auf eine speicherschonende Weise implementiert werden, ohne dabei einen wesentlichen Mehraufwand zur Überprüfung einzelner Tripel zu bedingen. Die so resultierende Tripel-Index-Struktur besteht entsprechend des Vertical-Partitioning aus mehreren Objekten (in Abhängigkeit der Anzahl der Prädikate), die jeweils mehrere Hash-Sets zur Speicherung von Tripeln mit einer komprimierten Bytegröße von 2 bis 16 Byte besitzen. Eine zusätzliche Speicherung der Tripel in einer Array-Struktur im Hauptspeicher des Rechners ist somit weder für die Auflösung von Referenzen der Working-Memories notwendig noch für die Erkennung von Duplikaten. Dadurch reduziert sich der permanent benötigte Hauptspeicher bei der Ausführung einer entsprechenden Reasoner-Implementierung auf den notwendigen Speicher für die beschriebene Tripel-Index-Struktur.

9.4 CODEGENERIERUNG FÜR DIE GPU

Die zuvor aufgeführten Konzepte tragen vor allem zu einer ressourcenschonenden Ausführung einer RETE-Implementierung unter Verwendung der GPU bei, wodurch das Ziel der Verarbeitung möglichst großer Datensätze auf einzelnen Rechnern verfolgt wird. Die Ausführungsgeschwindigkeit der einzelnen Arbeitsschritte, insbesondere bei der intensiven Nutzung der GPU, spielt aber eine ebenso große Rolle, um nicht nur die Möglichkeit zu bieten, große Datensätze zu verarbeiten, sondern dabei auch eine möglichst hohe Performance zu erreichen. Bei der Programmierung paralleler Hardware mit OpenCL existieren verschiedene Faktoren, die eine spätere Ausführungsgeschwindigkeit massiv beeinflussen können. Dazu zählen vor allem die Anzahl und die Reihenfolge der Zugriffe auf die verschiedenen Speicherbereiche, von denen der Zugriff auf den globalen Speicher am zeitaufwändigsten ist[2]. Ein weiterer Faktor ist das *Branching*, das durch Kontrollstrukturen wie Schleifen und Bedingungen auftreten kann und dazu führt, dass Threads innerhalb der SIMD-Architektur einen unterschiedlichen Ausführungspfad des Programmcodes durchlaufen und somit nicht die selben Instruktionen ausführen müssen. In dem Fall werden die unterschiedlichen Pfade innerhalb des Quellcodes sequentiell abgearbeitet, wodurch die tatsächlich erreichte Parallelität verringert wird.

Während weitere Faktoren, die die Ausführungsgeschwindigkeit beeinflussen (ein für die Berechnung auf der GPU geeignetes Problem vorausgesetzt), in der Fachliteratur nachgelesen werden können [Nei13] [GHK+12] [MGMG11], sind vor

2 Der Zugriff auf den globalen Speicher benötigt 600 bis 800 Taktzyklen, während bis zu acht Operationen innerhalb eines einzigen Taktzyklus auf dem lokalen Speicher ausgeführt werden können [Nvi09].

allem die beiden Aspekte des Speicherzugriffs und des Aufbaus der Logik durch Kontrollstrukturen abhängig vom jeweiligen Anwendungsfall und bieten damit ein hohes Optimierungspotential durch eine sorgfältig geplante Implementierung. Für eine bestmögliche Performance muss entsprechend die Anzahl der Speicherzugriffe auf den globalen Speicher ebenso minimiert werden[3] wie die Anzahl der möglichen Ausführungspfade innerhalb der Logik.

Insbesondere die Anzahl der möglichen Ausführungspfade wird bei der Implementierung eines regelbasierten Systems stark von der Variationsvielfalt der einzelnen Knoten beeinflusst. Für jeden Knoten können unterschiedliche Bedingungen gelten, die ggf. an verschiedenen Positionen eines Tripels bzw. eines Matches überprüft werden müssen. Das Alpha-Matching für das Pattern (?a rdf:type ?x) erfordert beispielsweise die Überprüfung des Prädikats, während das Pattern (?v rdf:type rdfs:Class) sowohl die Überprüfung des Prädikats als auch des Objekts auf einen bestimmten Wert vorsieht. Aufgrund der sich ergebenden Möglichkeiten sind für die Ausführung mehrere Schleifen notwendig, um während des Alpha-Matchings für alle Tripel sämtliche Alpha-Knoten zu durchlaufen und die jeweiligen Bedingungen zu überprüfen. Algorithmus 2 zeigt einen möglichen Aufbau der Logik, um das Alpha-Matching parallel auf der GPU unter Nutzung von einem Thread pro zu verarbeitendem Tripel auszuführen.

Zunächst müssen alle Tripel-Elemente aus dem globalen Speicher geladen werden. Anschließend erfolgt die Iteration durch alle Alpha-Knoten und die Überprüfung der für einen Knoten gegebenen Bedingungen. Nach der Iteration durch alle Bedingungen kann entschieden werden, ob für ein Tripel ein Eintrag in dem Working-Memory des jeweiligen Alpha-Knotens vorgenommen werden soll oder nicht.

Der zuvor aufgeführte Algorithmus zeigt, dass bereits für das Alpha-Matching auf der GPU trotz der grundlegend einfachen Operation (Vergleich von 1-3 Werten eines Tripels auf bestimmte Bedingungen), mehrere Kontrollstrukturen und Variablenzugriffe notwendig sind. Um die Ausführungslogik zu optimieren, kann durch die von der Gesamtlogik einer Reasoner-Implementierung separate Implementierung der Kernel ein generativer Ansatz etabliert werden, der den Code für die Ausführung auf der GPU zur Laufzeit bereitstellt. Der Schritt der Codegenerierung kann nach dem Dictionary-Encoding der Eingabedaten und Regeln erfolgen, so dass die jeweiligen numerischen Repräsentationen direkt im Programmcode für Vergleiche genutzt werden können und keine Übergabe der Bedingungen als Parameter beim Aufruf der Methoden notwendig sind. Algorithmus 3 zeigt ein vereinfachtes Beispiel der generierten Logik für das Alpha-Matching auf der GPU.

3 Beim Zugriff auf den Speicher ist nicht nur die Anzahl der Zugriffe relevant, sondern auch die Reihenfolge, in der auf verschiedene Elemente (z.B. in einem Array) zugegriffen wird.

Algorithmus 2 : Vereinfachter Algorithmus für das parallele Alpha-Matching

```
Data : alpha-nodes, conditions, triples
Result : working-memories

// read all triple elements from the global memory
read triple.subject for thread index
read triple.predicate for thread index
read triple.object for thread index

// iterate through all alpha-nodes
foreach node in alpha-nodes do
    node-conditions = read conditions for node
    matches = true
    // check all conditions for the alpha-node
    foreach condition in node-conditions do
        if triple does not match condition then
            matches = false
        end
    end
    if matches is true then
        add triple to working-memory of node
    end
end
```

Durch die Kenntnis über die zu verarbeitenden Regeln und der daraus abgeleiteten Alpha-Knoten kann zu Beginn der Ausführung des Alpha-Matchings gezielt entschieden werden, welche Tripel-Elemente tatsächlich benötigt werden und somit aus dem globalen Speicher geladen werden sollen. Im Anschluss daran folgt die Überprüfung der Bedingungen für die einzelnen Alpha-Knoten, ohne dabei zusätzliche Kontrollstrukturen wie z.B. Schleifen verwenden zu müssen. Des Weiteren müssen die einzelnen Tripel-Elemente nicht mit dem Inhalt von Variablen verglichen werden, sondern können direkt den im Quellcode berücksichtigten Werten gegenübergestellt werden.

Weiterhin können die Auswirkungen der unter Kapitel 7.3 aufgestellten Bedingung, dass $n_{p1} \geqslant n_{p2}$ ist, bei der Generierung der Kernel berücksichtigt werden. Das Ziel dieser Bedingung ist es, das Working-Memory des Elternknoten als Ausgangsbasis zu nutzen, das die meisten zu verarbeitenden Matches beinhaltet. Mit dieser Strategie wird vermieden, dass z.B. sehr wenige Threads durch eine Vielzahl an Elementen aus dem Working-Memory von p_2 iterieren müssen. Während p_1 und p_2 für das Beta-Matching zunächst beliebig vertauscht werden können, ist die Einhaltung einer festen Reihenfolge für das Erstellen der resultierenden Matches jedoch zwingend erforderlich. Zu diesem Zweck kann, basierend auf der generativen Vorgehensweise, einerseits eine Kernel-Implementierung für die Aus-

Algorithmus 3 : Algorithmus für das Alpha-Matching auf der GPU basierend auf den Eingabe-Regeln (generiert)

Data : triples
Result : working-memories

```
// read only needed triple elements from the global memory
read triple predicate for thread index
read triple object for thread index

// Check match for node-1
if triple predicate is equal to 517 then
  | add triple to working-memory of node1
end

// Check match for node-2
if triple predicate is equal to 37 and triple object is equal to 933 then
  | add triple to working-memory of node2
end

// Check match for further alpha-nodes
...
```

führung der Operation $(p_1 \ x \ p_2)$ bereitgestellt werden und andererseits für die Ausführung von $(p_2 \ x \ p_1)$.

Ebenso berücksichtigt werden kann die Einschränkung der Ausführungsumgebung, die eine Reservierung des verwendeten globalen Speichers bereits vor der Ausführung der Programmlogik bedingt. Diese Einschränkung führt dazu, dass sowohl das Alpha-Matching als auch das Beta-Matching in jeweils zwei Schritten ausgeführt werden müssen, wie es z.B. auch in [HP12] angewandt wird. Der erste Schritt dient der Berechnung des notwendigen Speichers, während der zweite Schritt (und damit die zweite Ausführung eines Kernels) die gleichen Operationen wiederholt, aber aufgrund des allokierten Speichers die Resultate für den Transfer zum Host bereitstellen kann.

Listing 5 zeigt einen generierten Programmcode für den ersten Schritt des Alpha-Matching, der die Anzahl der Matches für insgesamt fünf unterschiedliche Alpha-Pattern zählt.

Zusätzlich zu den bereits genannten Optimierungen kann für das Beta-Matching das Verfahren des *loop unrolling* [Sca12] angewandt werden. Anders als beim Alpha-Matching muss während des Beta-Matchings jeder Thread durch alle Einträge eines Working-Memories eines Eltern-Knotens iterieren. Die Codegenerierung eröffnet in diesem Fall die Möglichkeit, statt der Implementierung einer ein-

```
__kernel void alpha_match_count(    __global long *triples,
                                    __global int *match_counter,
                                    int max_gid) {
    // get the global id of this thread
    int gid = get_global_id(0);
    // initialize a local counter
    int local_matches = 0;

    /*
     * Skip this if there is nothing to do for this thread.
     * This might happen, if the number of triples is not evenly divisible
     * by the work-group size.
     */
    if(gid < max_gid) {

        // load the needed triple elements
        long triple_y = triples[gid*3+1];

        if(triple_y == 517) { local_matches += 3; }
        if(triple_y == 37) { local_matches += 3; }
        if(triple_y == 457) { local_matches += 3; }
        if(triple_y == 461) { local_matches += 3; }
        if(triple_y == 4) { local_matches += 3; }

        /*
         * store the number of matches found for this thread to the global
         * memory, so that it can be used in the next step to allocate
         * the needed memory to transfer back the matches, too.
         */
        match_counter[gid] = local_matches;
    }
}
```

Listing 5: Generierter Programmcode für das Alpha-Matching einer Regelbasis mit insgesamt fünf unterschiedlichen Alpha-Pattern

fachen Schleife, die während jeder Iteration genau einen Eintrag des Working-Memories prüft, eine Vielzahl dieser Prüfungen durch eine Wiederholung der Code-Fragmente (z.B. 32 Überprüfungen pro Iteration) innerhalb einer Schleife durchzuführen. Durch diese Vorgehensweise reduziert sich die Anzahl der notwendigen Überprüfungen der Schleifenbedingungen, wodurch ein weiterer Geschwindigkeitsvorteil erzielt werden kann.

Das vorgestellte Konzept der generativen Bereitstellung der Kernel während der Laufzeit kann aufgrund der zuvor aufgeführten Möglichkeiten sowohl für das Alpha-Matching als auch für das Beta-Matching und das Ableiten neuer Fakten auf der GPU angewandt werden, so dass sämtliche Kernel für die in dieser Arbeit vorgestellten Konzepte basierend auf den gegebenen Regeln generiert wer-

den können. Die Generierung erlaubt dabei im Allgemeinen die Optimierung der folgenden Aspekte:

- Reduzierung der notwendigen Übergabeparameter beim Aufruf von Kernel-Implementierungen auf der GPU
- Reduzierung der Speicher-Operationen auf dem globalen und lokalen Speicher, z.B. durch ausschließliches Laden der für den spezifischen Anwendungsfall notwendigen Informationen
- Reduzierung der notwendigen Kontrollstrukturen durch Integration der spezifischen Logik in den Programmcode
- Reduzierung des Mehraufwands bei der Verwendung von Schleifen durch *loop unrolling*.

10 EVALUATION

Das folgende Kapitel dient einerseits der Evaluation der erarbeiteten Konzepte und andererseits der Gegenüberstellung verwandter Arbeiten mit einem Schwerpunkt auf Aspekten der Ausführungsgeschwindigkeit und des Ressourcenbedarfs des Reasoning-Prozesses. Dazu wird nach einer kurzen Einführung in die verwendete Testumgebung, inklusive der genutzten Implementierung, eine Beschreibung und Dokumentation verschiedener Versuche vorgestellt. Die jeweiligen Versuche sind so aufgebaut, dass sie eine Beurteilung der unterschiedlichen Konzepte im Einzelnen erlauben und letztendlich deren Eignung für die Adressierung der in dieser Arbeit aufgestellten Forschungsfragen feststellen.

Teile der hier vorgestellten Ergebnisse wurden bereits in [PSZ15] veröffentlicht und werden zwecks einer ausführlichen Diskussion in dieser Arbeit erneut aufgegriffen. Zusätzlich finden sich in [PBSZ13b] und [PBSZ14] Ergebnisse zu einer Reasoner-Implementierung, die als Vorarbeit für die vorliegende Dissertation diente und im Rahmen dieser und der in [PSZ15] vorgestellten Arbeit weiterentwickelt wurde.

10.1 TESTUMGEBUNG

Die Testumgebung ergibt sich aus einer Implementierung, die die vorgestellten Konzepte zu einem regelbasiertem Reasoner für semantische Daten integriert und somit für das Reasoning auf verschiedenen Datensätzen unter Verwendung unterschiedlicher Regelsätze genutzt werden kann. Weiterhin werden zwei Hardwareumgebungen genutzt, um einerseits die Zielsetzung der Berechnung möglichst großer Datensätze auf einer einfachen Hardware zu evaluieren und andererseits gezielt die Auswirkungen einzelner Konzepte durch die Variation von Hardwarekonfigurationen zu betrachten.

10.1.1 *Implementierung*

Eine prototypische Implementierung der vorgestellten Konzepte wurde in Java vorgenommen. Ausschlaggebend für die Wahl dieser Sprache sind die Verfügbarkeit und der Verbreitungsgrad verschiedener Frameworks für die Implementierung semantischer Anwendungen, die ebenfalls in Java implementiert sind und somit eine mögliche Grundlage für die Nutzung der Reasoner-Implementierung

in verschiedenen Kontexten bieten. Zu den Frameworks zählen z.B. das *Jena*[1] Framework und die *OWL API*[2], die beide als Open Source veröffentlicht sind. Das Jena Framework wird in der prototypischen Reasoner-Implementierung genutzt, um sowohl RDF Daten als auch Regeln zu parsen. Für das Parsen von RDF Daten stehen verschiedene Implementierungen zur Verfügung, die es erlauben, unterschiedliche Eingabeformate wie z.B. RDF/XML[3], N-Tripel[4] oder Notation3[5] (N3) zu verarbeiten. Somit können die eigentliche Reasoner-Implementierung und die Umsetzung der vorgestellten Konzepte auf den im Jena-Framework intern verwendeten Datenstrukturen aufsetzen. Entsprechend erfolgt nach dem Parse-Vorgang eine Überführung der Jena-spezifischen Datenstrukturen in die für die Reasoner-Implementierung verwendeten Strukturen. Dabei wird zunächst anhand der geparsten Regeln ein RETE-Netzwerk abgeleitet, das die Struktur der weiteren Ausführung vorgibt. Ebenso erfolgt eine Überführung der RDF Daten in eine Dictionary-kodierte Repräsentation, der die in Kapitel 9.1 beschriebenen Datenstrukturen zugrunde liegen und die somit eine möglichst geringe Belastung des Hauptspeichers darstellen. Eine detaillierte Betrachtung des Dictionary-Encodings und der mit dem vorgestellten Konzept verbundenen Leistungsparameter findet in dem folgenden Kapitel 10.2.1 statt.

Nach dem Dictionary-Encoding der Regeln und der Tripel erfolgt die Generierung der Kernel (siehe Kapitel 9.4). Für die Ausführung der Kernel in einer OpenCL Umgebung wird zusätzlich die jocl-Bibliothek[6] als Java-Binding eingesetzt.

10.1.1.1 *Architekturkonzept und Realisierung der Host-Level-Parallelisierung*

Die Einführung der Host- und Device-Level-Parallelisierung (siehe Kapitel 7.4) erfordert neben der parallelen Ausführung von Logik auf der GPU ein Konzept, das die Host-seitige Parallelisierung unterstützt und damit einerseits zu einer besseren Auslastung von CPU-Ressourcen führt, aber andererseits auch zu einer optimalen Nutzung der verfügbaren Device-Kapazitäten beiträgt. Der konkrete Aufbau der dazu gewählten Architektur soll anhand eines einfachen RETE-Netzwerks erläutert werden, das in Abbildung 24 dargestellt ist. Das Netzwerk besteht aus vier Alpha-Knoten und zwei Beta-Knoten, die jeweils auf einer Ebene liegen. Somit lassen sich die Berechnungen für β_1 und β_2 parallel ausführen, was neben der Partitionierung der Arbeitslast für einzelne Knoten zusätzlich Teil einer möglichen Host-Level-Parallelisierung ist.

---

1 `https://jena.apache.org/`
2 `http://owlapi.sourceforge.net/`
3 `http://www.w3.org/TR/rdf-syntax-grammar/`
4 `http://www.w3.org/2001/sw/RDFCore/ntriples/`
5 `http://www.w3.org/TeamSubmission/n3/`
6 `http://jocl.org/`

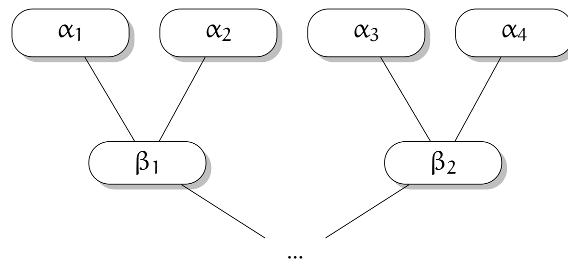


Abbildung 24: RETE-Netzwerk zur Veranschaulichung der Host-Level-Parallelisierung

Für die Realisierung der Parallelität innerhalb der Java-Anwendung werden Threads verwendet, deren Instanzen durch mehrere Thread-Pools verwaltet werden. Abbildung 25 zeigt einen schematischen Aufbau der Anwendung während des Beta-Matchings, der entsprechend des in Abbildung 24 skizzierten RETE-Netzwerks zwei Threads zur parallelen Bearbeitung der beiden Beta-Knoten verwendet. Jeder dieser Threads liest die notwendigen Matches der Elternknoten in Blöcken von der Festplatte (die Größe der Blöcke richtet sich nach der gewählten Größe für die Partitionierung der Arbeitslast), um sie zur Initialisierung eines Worker-Threads zu nutzen. Die dazu notwendigen Informationen werden einer synchronen Warteschlange (Queue) übergeben, die permanent von verfügbaren Worker-Threads abgefragt wird. Sobald ein Worker-Thread eine Aufgabe aus der Queue übernommen hat, ist er verantwortlich für die vollständige Ausführung der ihm übertragenen Aufgabe inklusive der Speicherung der jeweiligen Ergebnisse.

Ein initialisierter Worker-Thread übergibt sich selber inklusive aller für die Ausführung der gewählten Operation notwendigen Daten einer weiteren Queue (GPU-Queue), deren Worker für die Ausführung von Operationen auf massiv paralleler Hardware (GPUs) zuständig sind. Um den exklusiven Zugriff auf jeweils eine GPU durch einen Thread sicherzustellen, wird pro verfügbarer GPU innerhalb eines Systems exakt ein *GPU-Worker* (Thread) erzeugt. Sobald ein GPU-Worker eine Aufgabe von der GPU-Queue entgegennimmt, werden die definierten Operationen ausgeführt und die Ergebnisse der Ausführung von der GPU gelesen. Die weitere Verarbeitung der Ergebnisse, die z.B. die Speicherung neuer Matches beinhalten kann, obliegt anschließend wieder dem Thread aus dem Worker-Pool, so dass der GPU-Worker direkt mit der Ausführung einer weiteren Operation aus der GPU-Queue fortfahren kann. Ebenso wie die in Abbildung 25 für das Beta-Matching beschriebene Vorgehensweise zur Anwendung des Konzepts der Host-Level-Parallelisierung lässt sich das Konzept auch für das Alpha-Matching sowie für das Ableiten neuer Fakten auf der GPU anwenden. Während für das Alpha-Matching lediglich ein Thread für die Bereitstellung der Arbeitslast über die Worker-Queue verantwortlich ist, kann für das Ableiten von Fakten ein Thread

pro zu feuernder Regeln verwendet werden, um entsprechende Worker-Threads zu initialisieren.

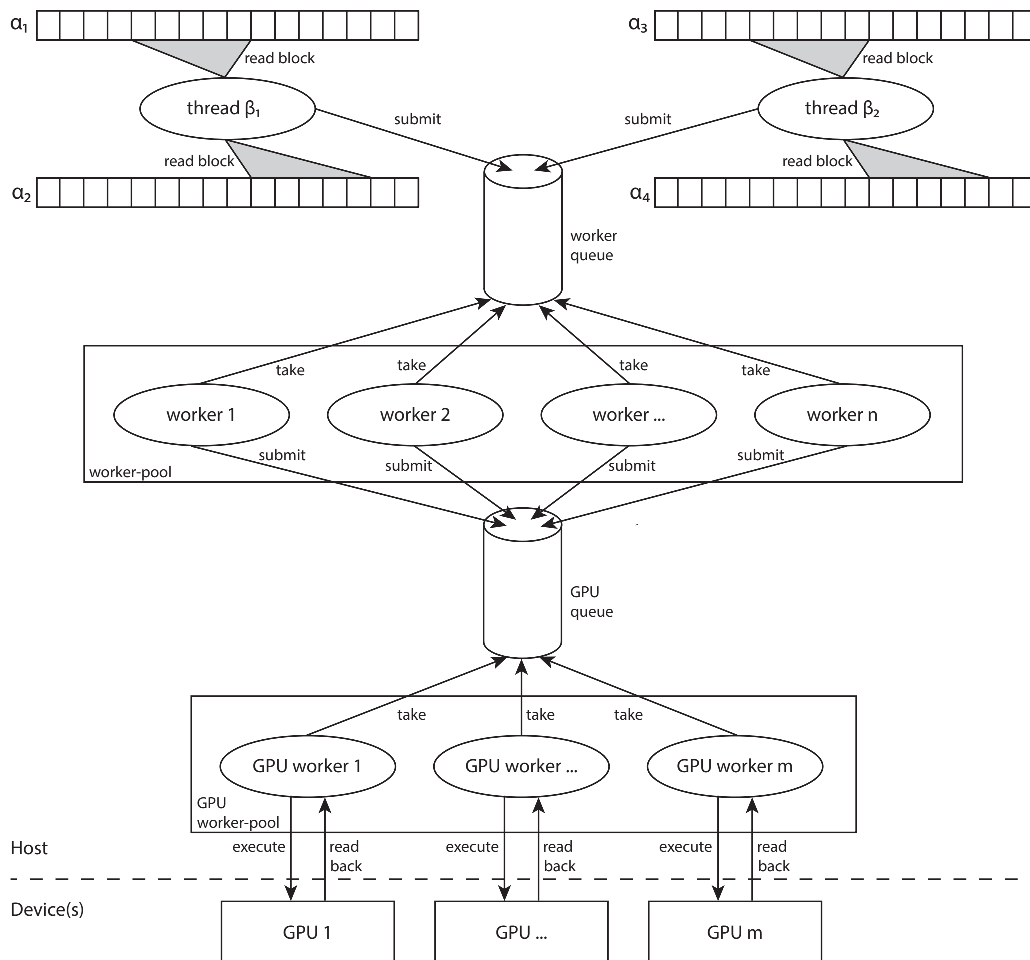


Abbildung 25: Architekturkonzept der Reasoner-Implementierung zur Umsetzung der Host-Level-Parallelisierung am Beispiel des Beta-Matchings für das RETE-Netzwerk aus Abbildung 24

.

Durch das vorgestellte Konzept zur Umsetzung der Host-Level-Parallelisierung unter Verwendung mehrerer Thread-Pools werden verschiedene Ziele verfolgt. Der Einsatz der Worker-Queue erlaubt den jeweiligen Threads für die Beta-Knoten einen sequenziellen Lesezugriff auf einzelne Dateien, um die Matches blockweise und entsprechend ihrer Reihenfolge zu lesen und einen unkoordinierten und damit auch zeitintensiveren [BC07] Zugriff durch mehrere Threads zu vermeiden. Die Synchronität der Worker-Queue hingegen sorgt dafür, dass nicht beliebig viele Partitionen der Arbeitslast vorbereitet werden können, ohne dass entsprechende

Worker-Threads zur Verfügung stehen. Andernfalls könnte durch diesen Mechanismus durch das Auslesen der jeweils notwendigen Informationen von der Festplatte der Hauptspeicher unnötig belastet werden. Die GPU-Queue und die GPU-Worker tragen hingegen dazu bei, einerseits einen exklusiven Zugriff auf eine physikalische GPU sicherzustellen und andererseits eine bestmögliche Auslastung der Hardware zu erzielen. Durch die ausschließliche Ausführung von Operationen auf der GPU durch die GPU-Worker und der Überlassung der Weiterverarbeitung der Ergebnisse an den aufrufenden Worker-Thread ist sichergestellt, dass keine unnötigen Pausen bei der Verwendung der verfügbaren GPU-Ressourcen durch die Verarbeitung von Ergebnissen entstehen.

Zusätzlich zu den bereits aufgeführten Vorteilen der Anwendungsarchitektur erlaubt die Verwendung von Thread-Pools die Wiederverwendung von bereits bestehenden Objekten, was sich durch eine effizientere Bereitstellung bemerkbar macht, aber auch den Aufwand zur Speicherverwaltung durch den Garbage Collector der Java Umgebung reduziert. Entsprechend stammen beispielsweise auch die Threads, die in Abbildung 25 die Threads für β_1 und β_2 darstellen, aus einem weiteren, allgemeinen Thread-Pool, der zugunsten der Übersichtlichkeit nicht in die Darstellung mit aufgenommen wurde. Des Weiteren wird ein ähnliches Konzept für die Verwendung von Arrays angewandt, die der Speicherung von Tripeln und Matches in einer numerischen 64 Bit Repräsentation (sämtliche Dictionary-kodierte Werte werden als 64 Bit Long-Wert gespeichert) dienen. Statt der permanenten Erstellung neuer Arrays, beispielsweise beim Lesen der Matches von der Festplatte, werden auch diese gepoolt und wiederverwendet, um den Ressourcenverbrauch und die Belastung des Speichermanagements zu reduzieren.

10.1.1.2 *Programmausführung*

Die Ausführung des Reasoner-Prozesses erlaubt entsprechend der eingeführten Konzepte eine Konfiguration verschiedener Parameter zur Anpassung an eine gegebene Hardwareumgebung. Während diese zum Teil bereits in den vorherigen Kapiteln als Möglichkeit der Adaption an eine bestimmte Ausführungsumgebung aufgeführt wurden, werden hier noch einmal die wichtigsten Parameter zusammengefasst:

CACHE-GRÖSSE: Während des Dictionary-Encodings wird ein Cache verwendet, der bei wiederkehrenden Tripel-Elementen einen aufwändigen Lese-Vorgang auf der Festplatte vermeiden soll. Entsprechend des verwendeten Datensatzes und der damit verbundenen erwarteten Heterogenität der Tripel und dem verfügbaren Hauptspeicher kann die Größe des Caches vor der Ausführung festgelegt werden.

THREAD-POOL-GRÖSSE: Die Thread-Pool-Größe definiert die Anzahl der Worker-Threads, die für den Worker-Pool bereitgestellt werden. Die Größe sollte in Abhängigkeit einer Gesamt-Betrachtung der verwendeten Hardware und nach Möglichkeit auch der verwendeten Daten (RDF-Daten und Regeln) erfolgen. Zu berücksichtigen sind die Anzahl der verfügbaren CPU-Kerne, aber auch Faktoren wie die Anzahl der verwendeten GPUs und der mögliche Festplattendurchsatz sind relevant und sollten mit der erwarteten Rechenbelastung durch die Eingabedaten in ein abgestimmtes Verhältnis gebracht werden. Eine zu hohe Anzahl an Threads kann neben einem erhöhten Speicherverbrauch zu einem größeren Scheduling-Aufwand auf der CPU, aber auch zu langen Wartezeiten bei exklusiven Zugriffen wie dem Speichern von Ergebnissen führen. Eine zu geringe Pool-Größe hingegen lässt ggf. Ressourcen ungenutzt und führt so zu einem Performanceverlust.

Zur Annäherung an eine geeignete Größe kann die Thread-Pool-Größe zunächst entsprechend der Anzahl der verfügbaren CPU-Kerne gewählt werden. Bei der Verwendung eines Regelsatzes in Kombination mit einem Datensatz, bei dem nur eine sehr geringe Rechenlast entsteht (besonders beim Beta-Matching), kann die Anzahl Schrittweise erhöht werden, um einen maximalen Datenfluss zu erreichen.

(G_x, G_y) ALPHA-MATCHING: Der durch (G_x, G_y) aufgespannte globale Indexraum (siehe Kapitel 6.3) legt während des Alpha-Matchings die Größe der Partitionen fest, in die die Eingabetripel und die zu berücksichtigen Alpha-Knoten geteilt werden. G_x definiert dabei die Anzahl der Tripel und sollte so gewählt werden, dass eine entsprechende Anzahl an Tripel in den globalen Speicher der verwendeten GPU geladen werden können und sich die Anzahl der Tripel durch die verwendete work-group-Größe teilen lässt. Die Teilbarkeit von G_x durch die work-group-Größe stellt eine optimale Nutzung der verfügbaren Hardwareressourcen sicher. G_y hingegen wird durch die Anzahl der Alpha-Knoten definiert und muss nicht explizit gewählt werden.

Beispielsweise kann für einen Datensatz mit mindestens mehreren zehn Millionen Tripeln ein G_x von $2^{21} = 2.097.152$ gewählt werden. Der so aufgespannte Indexraum lässt sich durch Zweierpotenzen dividieren (und somit auch durch die work-group-Größe der verwendeten GPU) und führt mit rund 6,3 MB Speicherverbrauch für die Tripel zu einer schnellen Übertragung und Verarbeitung.

(G_x, G_y) BETA-MATCHING: Die Wahl der Größe des verwendeten Indexraums während des Beta-Matchings gestaltet sich ähnlich wie beim Alpha-Matching. Ein Unterschied besteht jedoch in der Wahl von G_y, das explizit angegeben

werden muss und bei der Bestimmung der Partitionsgröße die Anzahl der zu verwendenden Matches von p_2 definiert.

Anders als G_x sollte G_y kleiner gewählt werden, um lange laufende Operationen auf der GPU zu vermeiden und einen kontinuierlichen Datenfluss zu ermöglichen (schließt das Lesen und Schreiben der benötigten bzw. erzeugten Daten mit ein). Als eine optimale Größe hat sich auf unterschiedlicher Hardware 20.480 herausgestellt. Um die Auswirkungen dieser Größe in Abhängigkeit der konkreten Hardware- und Datenvoraussetzungen zu testen, bietet sich ein Test mit jeweils einer Verdoppelung bzw. Halbierung dieses Wertes für G_y an, um sich so einem Optimalwert zu nähern.

(G_x, G_y) FÜR DAS ABLEITEN NEUER FAKTEN: Der globale Indexraum für das Ableiten neuer Fakten auf der GPU definiert sich ausschließlich über G_x, das die Anzahl der abzuleitenden Fakten pro zu verarbeitender Partition angibt. Die Wahl der Größe muss unter Berücksichtigung des verfügbaren Speichers der verwendeten GPU(s) gewählt werden und sollte ebenfalls durch die work-group-Größe teilbar sein.

Während ein G_x, das für eine weitestgehende Auslastung des globalen GPU-Speichers sorgt, nicht zwangsweise zu einer längeren Gesamtlaufzeit der GPU-Operationen führt, wird dennoch der Datenfluss innerhalb der Anwendung unterbrochen. Das macht sich insbesondere dadurch bemerkbar, dass z.B. die rechenintensive Operation der Deduplikation im Anschluss für alle erzeugten Tripel gleichzeitig ausgeführt werden muss. Aus diesem Grund hat sich für G_x eine Größe von 524.288 (2^{19}) bzw. 819.200 (Teilbar durch 1024) als praktikabel erwiesen, so dass einerseits eine hohe Parallelisierung bei der Ableitung der Tripel stattfinden kann, aber gleichzeitig auch die Lese-, Schreib- und Validierungsschritte kontinuierlich auf Teildaten ausgeführt werden können.

Zusätzlich zu den jeweils spezifischen Faktoren für die Wahl des globalen Indexraums für die verschiedenen Schritte des RETE-Algorithmus, muss im Allgemeinen neben der Berücksichtigung der Speicherbelastung auf der GPU auch eine Betrachtung der jeweiligen Ausführungszeit erfolgen. Nur wenn die Ausführung einzelner Operationen auf der massiv parallelen Hardware nicht zu einer Blockierung der Host-Level-Parallelisierung führt, kann ein kontinuierlicher Datenfluss und damit ein hoher Durchsatz erzielt werden.

10.1.2 *Verwendete Datensätze*

Für die Evaluation der beschriebenen Konzepte werden drei unterschiedliche Datensätze herangezogen.

Ein erster Datensatz, der für verschiedene Versuchsdurchführungen genutzt wird, ist der *Lehigh University Benchmark* [GPH05], bei dem es sich um einen künstlichen Datensatz handelt, der mithilfe eines Generators erstellt werden kann. Der Generator erlaubt die Bereitstellung beliebig vieler Universitäts-Beschreibungen, wodurch sich die Anzahl der zu erstellenden Tripel skalieren lässt. Entsprechend kennzeichnet im Folgenden beispielsweise die Bezeichnung LUBM1000 den Lehigh University Benchmark-Datensatz mit 1000 generierten Universitäten, während LUBM2000 exakt 2000 generierte Universitäten referenziert usw. Die Wahl des LUBM-Benchmarks stützt sich auf mehrere Faktoren. Zum einen eignet er sich durch die Skalierbarkeit für Evaluationen, die das Verhalten eines Systems unter einer gleichmäßig steigenden Belastung zeigen sollen. Zum anderen lassen sich Datensätze mit einer bestimmten Zielgröße erstellen, um Maximalbelastungen besser evaluieren zu können. Des Weiteren wurde der Benchmark bereits von einer Vielzahl verwandter Arbeiten verwendet (z.B. [UKOH09] [UKM+12] [WH09] [UMJ+13] [SP08b] [WED+08]), wodurch sich eine einfache Vergleichbarkeit für die Ausführungsgeschwindigkeit ergibt.

Zusätzlich zum künstlichen Datensatz werden zwei Datensätze verwendet, die aus einem realen Anwendungskontext stammen. *DBpedia* [LIJ+14] ist ein Gemeinschaftsprojekt verschiedener Universitäten und Forschungseinrichtungen und stellt Daten, die aus den strukturierten Informationen der Wikipedia-Seite extrahiert werden, im RDF Format frei zur Verfügung, um so den Aufbau semantischer Anwendungen, basierend auf einer sehr heterogenen, aber auch umfassenden Datenbasis, zu ermöglichen. Diese Heterogenität zeigt sich beispielsweise auch in der in Kapitel 2.5 gezeigten Linked Open Data Cloud, in der DBpedia nicht nur den größten Datensatz darstellt, sondern auch den mit den meisten Links zu anderen Datensätzen. DBpedia gliedert sich in Abhängigkeit der Sprache in unterschiedliche Datensätze, die jeweils aus den länderspezifischen Wikipedia Seiten extrahiert wurden und entsprechend auch einen unterschiedlichen Umfang aufweisen können. Neben der DBpedia-Ontologie werden für diese Arbeit alle zum August 2014 verfügbaren Datensätze der englischen Sprache verwendet, die auch bereits in [PSZ15] benutzt wurden und insgesamt rund 451 Millionen Fakten (rund 394 Millionen eindeutige Fakten) umfassen.

Als weiterer Datensatz wird der aus der Domäne der Biomedizin stammende Datensatz *Comparative Toxicogenomics Database* [MRD+06] (CTD) verwendet, der Informationen im Bereich der Toxikogenomik beinhaltet. Diese beschreiben mit rund 641 Millionen Fakten (Datensatz vom Juni 2014, beinhaltet rund 335 Millionen eindeutige Fakten) beispielsweise Beziehungen zwischen Chemikalien, genetischen Gegebenheiten und Krankheiten und unterstützen die Erforschung und das Verständnis der Auswirkungen von Umweltchemikalien auf die menschliche Gesundheit. Im Gegensatz zu DBpedia ist der CTD Datensatz sehr domänenspezi-

fisch und deshalb weniger heterogen, wodurch neben dem Benchmark-Datensatz zwei strukturell unterschiedliche Datensätze für die Evaluation genutzt werden.

10.1.3 *Verwendete Regelsätze*

Für die Evaluation werden insgesamt drei Regelsätze verwendet, die insbesondere für das regelbasierte Reasoning auf großen Datensätzen weit verbreitet sind und auch von verschiedenen MapReduce basierten Ansätzen implementiert werden [UKOH09] [UKM+12] [WH09]. RDF-Schema (RDFS) wurde in Kapitel 2.3 bereits als vom W3C spezifizierte Beschreibungssprache eingeführt. Für das Reasoning mit RDFS sind insgesamt 13 Regeln definiert, die durch einen forward-chaining-Reasoner angewandt werden müssen und in Tabelle 1 aufgelistet sind.

Name	if	then
rdf1	(?x ?p ?y)	(?p rdf:type rdf:Property)
rdfs2 *	(?p rdfs:domain ?c), (?x ?p ?y)	(?x rdf:type ?c)
rdfs3 *	(?p rdfs:range ?c), (?x ?p ?y)	(?y rdf:type ?c)
rdfs4a	(?x ?p ?y)	(?x rdf:type rdfs:Resource)
rdfs4b	(?x ?p ?y)	(?y rdf:type rdfs:Resource)
rdfs5 *	(?p rdfs:subPropertyOf ?q), (?q rdfs:subPropertyOf ?r)	(?p rdfs:subPropertyOf ?r)
rdfs6	(?p rdf:type rdf:Property)	(?p rdfs:subPropertyOf ?p)
rdfs7 *	(?p rdfs:subPropertyOf ?q), (?x ?p ?y)	(?x ?q ?y)
rdfs8	(?c rdf:type rdfs:Class)	(?c rdfs:subClassOf rdfs:Resource)
rdfs9 *	(?c rdfs:subClassOf ?d), (?x rdf:type ?c)	(?x rdf:type ?d)
rdfs10	(?c rdf:type rdfs:Class)	(?c rdfs:subClassOf ?c)
rdfs11 *	(?c rdfs:subClassOf ?d), (?d rdfs:subClassOf ?e)	(?c rdfs:subClassOf ?e)
rdfs12	(?p rdf:type rdfs:ContainerMembershipProperty)	(?p rdfs:subPropertyOf rdfs:member)
rdfs13	(?x rdf:type rdfs:Datatype)	(?x rdfs:subClassOf rdfs:Literal)

Tabelle 1: RDF/RDFS Ableitungsregeln

Von den 13 RDFS Regeln besitzen insgesamt sechs Regeln genau zwei Regel-Terme (diese sind in Tabelle 1 zusätzlich mit einem * gekennzeichnet), die gemeinsam das RDFS Subset ρdf [MPG07] bilden. ρdf eignet sich insbesondere, um die Komplexität des Reasoning-Vorgangs weiter zu reduzieren und bei großen Datensätzen einen höheren Durchsatz zu erreichen, aber gleichzeitig die wesentliche Semantik zu erhalten [MPG07]. Die ausgelassenen Regeln der RDFS Semantik, die ausschließlich aus einem einzigen Regel-Term bestehen, können bei Bedarf während der Abfrage eines entsprechenden Repositories direkt auf die Antwort-Informationen angewandt werden [UKOH09].

Als dritter Regelsatz wird pD* [tH05] verwendet, der sowohl RDFS- als auch OWL-Konstrukte beinhaltet und damit auch das OWL-Reasoning adressiert. Der

Name	if	then
rdfp1	(?p rdf:type owl:FunctionalProperty), (?u ?p ?v), (?u ?p ?w)	(?v owl:sameAs ?w)
rdfs2	(?p rdf:type owl:InverseFunctionalProperty), (?v ?p ?u), (?w ?p ?u)	(?v owl:sameAs ?w)
rdfp3	(?p rdf:type owl:SymmetricProperty), (?v ?p ?u)	(?u ?p ?v)
rdfp4	(?p rdf:type owl:TransitiveProperty), (?u ?p ?w), (?w ?p ?v)	(?u ?p ?v)
rdfp5a	(?u ?p ?v)	(?u owl:sameAs ?u)
rdfp5b	(?u ?p ?v)	(?v owl:sameAs ?v)
rdfp6	(?v owl:sameAs ?w)	(?w owl:sameAs ?v)
rdfp7	(?v owl:sameAs ?w), (?w owl:sameAs ?u)	(?v owl:sameAs ?u)
rdfp8ax	(?p owl:inverseOf ?q), (?v ?p ?w)	(?w ?q ?v)
rdfp8bx	(?p owl:inverseOf ?q), (?v ?q ?w)	(?w ?p ?v)
rdfp9	(?v rdf:type owl:Class), (?v owl:sameAs ?w)	(?v rdfs:subClassOf ?w)
rdfp10	(?p rdf:type owl:Property), (?p owl:sameAs ?q)	(?p rdfs:subPropertyOf ?q)
rdfp11	(?u ?p ?v), (?u owl:sameAs ?x), (?v owl:sameAs ?y)	(?x ?p ?y)
rdfp12a	(?v owl:equivalentClass ?w)	(?v rdfs:subClassOf ?w)
rdfp12b	(?v owl:equivalentClass ?w)	(?w rdfs:subClassOf ?v)
rdfp12c	(?v rdfs:subClassOf ?w), (?w rdfs:subClassOf ?v)	(?v rdfs:equivalentClass ?w)
rdfp13a	(?v owl:equivalentProperty ?w)	(?v rdfs:subPropertyOf ?w)
rdfp13b	(?v owl:equivalentProperty ?w)	(?w rdfs:subPropertyOf ?v)
rdfp13c	(?v rdfs:subPropertyOf ?w), (?w rdfs:subPropertyOf ?v)	(?v rdfs:equivalentProperty ?w)
rdfp14a	(?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u ?p ?v)	(?u rdf:type ?v)
rdfp14bx	(?v owl:hasValue ?w), (?v owl:onProperty ?p), (?u rdf:type ?v)	(?u ?p ?v)
rdfp15	(?v owl:someValuesFrom ?w), (?v owl:onProperty ?p), (?u ?p ?x), (?x rdf:type ?w)	(?u rdf:type ?v)
rdfp16	(?v owl:allValuesFrom ?u), (?v owl:onProperty ?p), (?w rdf:type ?v), (?w ?p ?x)	(?x rdf:type ?u)

Tabelle 2: pD* Ableitungsregeln [tH05]

Regelsatz besteht aus insgesamt 16 Regeln, die aus bis zu vier Regel-Termen bestehen und in Tabelle 2 dargestellt sind. In Bezug auf das Reasoning mittels einer RETE-Implementierung stellt der pD* Regelsatz vor allem aufgrund der Vielzahl an Regeln mit mehreren Regel-Termen eine höhere Komplexität als die zuvor beschriebenen Regelsätze dar. Dazu tragen vor allem die Regeln `rdfp15` und `rdfp16` bei, die nicht nur aus jeweils vier Regel-Termen bestehen, sondern deren Regel-Terme zum Teil auch sehr unspezifisch bzw. allgemein sind und damit zu einer Vielzahl an positiven Evaluationen führen können.

10.1.4 *Hardwareumgebung*

Als Ausführungsumgebung für die Evaluation werden zwei verschiedene Hardwarekonfigurationen verwendet. Zum einen wird ein MacBook Pro Laptop (Retina, Mitte 2012) mit einem 2,3 GHz Intel Core i7 Prozessor, einer 256 GB SSD Festplatte, 16 GB Arbeitsspeicher und einer NVIDIA GeForce GT 650M mit 1024 MB Speicher

verwendet. Das MacBook dient der Betrachtung der Möglichkeit des Reasonings über große Datenmengen mit einer Hardware, die ohne besondere Anforderungen auskommt. Im Gegensatz zu vielen verwandten Arbeiten [UKOH09] [UKM+12] [WH09] [LQWY11] [ZQL+12], die dem Reasoning über große Datenmengen mit einem MapReduce basierten Ansatz begegnen und damit auf eine nahezu beliebig skalierbare Hardware zurückgreifen, kann die Hardware des Laptops nicht beliebig skaliert werden, stellt aber gleichzeitig eine häufig verfügbare und verhältnismäßig kostengünstige Hardwareumgebung dar.

Um bei der Evaluation der verschiedenen Konzepte Variationsmöglichkeiten bezüglich der verwendeten Ressourcen nutzen zu können und gleichzeitig mehr Möglichkeiten bei der Verarbeitung großer Datenmengen als beim MacBook zu eröffnen, wird zusätzlich eine Workstation mit einem 2GHz Intel Xeon Prozessor mit 6 Kernen, 64 GB Arbeitsspeicher und insgesamt drei SSD Festplatten (128 GB, 256 GB und 500 GB) verwendet. Die verschiedenen Festplatten werden dazu genutzt, um die Tripel, die Alpha-Matches und die Beta-Matches auf verschiedenen Laufwerken zu speichern und so einen höheren Durchsatz zu erzielen. Beispielsweise werden so für das Alpha-Matching die Tripel von einem Laufwerk gelesen, während gleichzeitig Ergebnisse auf ein anderes Laufwerk geschrieben werden können. Als weitere Ausstattung sind in der Workstation zwei AMD Radeon HD 7970 GHz Edition Gaming Grafikkarten der Mittelklasse mit jeweils 3 GB Speicher verbaut. Als Betriebssystem dient ein Ubuntu Server 14.04.

10.1.5 *Bestimmung des notwendigen Speicherplatzes einer naiven Implementierung*

Bevor die vorgestellten Konzepte zur Reduzierung des notwendigen Speicherplatzes evaluiert werden, soll zunächst der Speicherberdarf einer naiven Implementierung festgestellt werden, um in den folgenden Abschnitten eine Möglichkeit des Vergleichs zu schaffen. Unter einer *naiven* Implementierung wird in dieser Arbeit ein Lösungsansatz verstanden, der ohne den gezielten Einsatz von Konzepten zur Reduzierung des Speicherbedarfs auskommt. Insbesondere folgt daraus die Speicherung sämtlicher Tripel in einer Array-Struktur. Die Array-Struktur dient wiederum der Auflösung der Referenzen, die in den Working-Memories der Alpha- und Beta-Knoten gespeichert werden, sowie der Überprüfung auf Duplikate. Die Überprüfung auf Duplikate bedingt in einer minimalen Implementierung mindestens ein HashSet, dessen Werte jeweils auf die Position innerhalb des Tripel-Arrays zeigen. Als weitere Struktur ist das Dictionary zur Rücktransformation der Tripel aus der numerischen Repräsentation in die ursprüngliche Darstellung notwendig, dessen Einträge ebenfalls im Hauptspeicher vorgehalten werden müssen. Tabelle 3 zeigt den aus den zuvor aufgeführten Datenstrukturen resultierenden Speicherbedarf für zwei Datensätze, nachdem die ρdf und RDFS Regeln angewandt wurden.

Datensatz	Regelsatz	Tripel (n)	Tripel-größe	Tripel HashSet	Referenzen in W	Größe von W	Einträge Dictionary	Größe Dictionary	Gesamt-größe
LUBM2000	ρdf	333,7 M	8.009 MB	3.814 MB	1.022,5 M	8.180 MB	65,8 M	4.043 MB	24,0 GB
LUBM2000	RDFS	377,1 M	9.050 MB	4.310 MB	2.119,1 M	16.953 MB	65,8 M	4.043 MB	34,4 GB
DBpedia	ρdf	400,6 M	9.614 MB	4.578 MB	446,7 M	3.574 MB	127,7 M	9.185 MB	27,0 GB
DBpedia	RDFS	475,1 M	11.402 MB	5.430 MB	1.978,5 M	15.828 MB	127,7 M	9.185 MB	41,8 GB

Tabelle 3: Speicherverbrauch der notwendigen Datenstrukturen für eine naive Reasoner-Implementierung

Für die Berechnung des Speicherverbrauchs der HashSets wurde ein Load-Faktor von 0,7 angenommen. Die Referenzen beschreiben die Anzahl der Einträge innerhalb der Working-Memories und wurden ebenso wie die Größe des Dictionaries durch die Ausführung des RETE-Algorithmus auf den jeweiligen Datensätzen ermittelt.

Die Bestimmung des notwendigen Speicherplatzes in Tabelle 3 zeigt, dass bereits beim Reasoning auf Datensätzen mit 300 bis 500 Millionen Fakten ein Hauptspeicher von über 40 GB notwendig sein kann, um die grundlegenden Datenstrukturen abzulegen. Ein Ziel der vorliegenden Arbeit ist es, diesen Speicherbedarf zu reduzieren, um auch Large-scale Reasoning auf einfacher Hardware zu ermöglichen.

10.2 VERSUCHSDURCHFÜHRUNG

Die verwendeten Konfigurationsparameter für die Ausführung der Versuche variieren je nach Hardwareumgebung leicht und sind in Tabelle 4 dargestellt. Die für G_x gewählten Werte sind jeweils ein Vielfaches der maximalen Work-Group-Größe der verwendeten Hardware (1024 bei dem MacBook und der NVIDIA GeForce GT 650M und 256 für die Workstation mit der AMD Radeon HD 7970). Jeder Versuch wird insgesamt fünfmal ausgeführt, während die mittlere Ausführungszeit angegeben wird.

Parameter	MacBook	Workstation
Thread-Pool-Größe:	6	8
(G_x) Alpha-Matching:	2.097.152	2.097.152
(G_x, G_y) Beta-Matching:	2.097.152 x 20.480	2.097.152 x 20.480
(G_x) für das Ableiten von Fakten:	524.288	819.200

Tabelle 4: Konfigurationsparameter der verwendeten Hardwareumgebungen

10.2.1 *Dictionary-Encoding unter beschränkten Ressourcen*

Das Konzept des Dictionary-Encodings unter Verwendung von Festplattenspeicher wurde nicht nur eingeführt (Kapitel 9.1.2), um auch unter beschränkten Ressourcen große Datensätze parsen und kodieren zu können, sondern auch, um den Speicherbedarf während des Reasoning-Prozesses zu minimieren. Die Minimierung des Speicherbedarfs nach dem eigentlichen Kodierungsvorgang ergibt sich bereits aus dem in Kapitel 9.1.2 eingeführten Konzept, das eine vollständige Auslagerung der Informationen auf die Festplatte ermöglicht. Die folgende Evaluation zeigt zusätzlich die Leistungsparameter des Kodierungs- und Dekodierungs-Vorgangs, die bei der Ausführung auf unterschiedlichen Datensätzen erreicht werden. Um der Ausrichtung der vorliegenden Arbeit auf eine Hardware mit beschränkten Ressourcen Rechnung zu tragen, wurde das MacBook als Ausführungsumgebung gewählt, dem lediglich 16 GB Hauptspeicher zur Verfügung stehen. Als Datensätze wurden sowohl verschiedene Größen des LUBM-Datensatzes gewählt als auch DBpedia und CTD. Für den Vorgang des Parsens der Dateien wird das Jena-Framework verwendet, das die Grundlage für den Prozess des Dictionary-Encodings bildet. Um das Parsen zu parallelisieren, werden mehrere Parser-Threads erstellt, die die jeweils zu parsenden Dateien über eine Queue entgegennehmen und abarbeiten. Für diese Vorgehensweise eignet sich besonders der Benchmark-Datensatz, der aus mehreren einzelnen Dateien pro generierter Universität besteht und damit eine gute Möglichkeit der parallelen Verarbeitung bietet. Die Menge der Dateien für DBpedia und CTD hingegen belaufen sich auf eine deutlich geringere Anzahl, deren Größe stark variiert, weshalb eine optimale Auslastung der verfügbaren Ressourcen nicht sichergestellt ist. Tabelle 5 zeigt die Ergebnisse für das Dictionary-Encoding, wobei der Durchsatz in *kilo Tripel pro Sekunde* (ktps) angegeben wird und somit die Anzahl der kodierten bzw. dekodierten Tripel pro Sekunde beschreibt.

In Abhängigkeit des Datensatzes wurde eine unterschiedliche Cache-Größe gewählt. Insbesondere der Benchmark-Datensatz besteht aus Tripel-Fragmenten, die sich entweder stets wiederholen oder durch die Beschreibung neuer Instanzen noch nicht im Datenbestand existieren. Aus diesem Grund ist bereits ein kleiner Cache ausreichend, um eine hohe Trefferquote zu erzielen. Der DBpedia-Datensatz hingegen ist sehr heterogen und sich wiederholende Tripel-Fragmente liegen in den zu verarbeitenden Dokumenten zum Teil weiter auseinander, was der Grund für die Wahl eines größeren Caches ist. Die Eigenschaften des Datensatzes machen sich auch im Durchsatz für das Kodieren bzw. Dekodieren bemerkbar. Während für die Benchmark-Datensätze und den CTD-Datensatz ein durchschnittlicher Durchsatz von rund 187 ktps für die Kodierung erzielt wird, liegt der Durchsatz für DBpedia bei lediglich 23,1 ktps. Die Dekodierung der Datensätze,

Datensatz	Input Tripel	eindeutige Tripel	Cache-größe	Laufzeit		Durchsatz (ktps)	
				Kodierung	Dekod.	Kodierung	Dekod.
LUBM1000	138,3 M	133,6 M	500 k	703 s	73 s	196,9 ktps	1.841,5 ktps
LUBM2000	276,4 M	267,0 M	500 k	1.400 s	157 s	197,5 ktps	1.700,7 ktps
LUBM4000	553,0 M	534,2 M	500 k	2.809 s	351 s	196,9 ktps	1.841,5 ktps
LUBM8000	1.106,0 M	1.068,4 M	500 k	6.777 s	736 s	161,6 ktps	1.457,3 ktps
DBpedia	450,5 M	393,6 M	25.000 k	19.478 s	3036 s	23,1 ktps	129,7 ktps
CTD	640,6 M	324,4 M	5.000 k	3.528 s	458 s	181,8 ktps	710,3 ktps

Tabelle 5: Ergebnisse des Dictionary-Encodings [PSZ15]

für die über die numerische Repräsentation der Werte direkt auf den entsprechenden Eintrag im Dictionary zugegriffen werden kann, wird ein Durchsatz von rund 130 ktps für DBpedia und bis zu 1457 ktps für den Benchmark-Datensatz erreicht.

Mit der Durchführung der Tests konnte gezeigt werden, dass das Dictionary-Encoding von Datensätzen mit bis zu 1,1 Milliarden Tripeln auf einem einzelnen Laptop möglich ist. Maßgeblich hierfür ist die Auslagerung der Dictionary-Einträge auf die Festplatte, die neben den reinen Daten (z.B. ca. 9 GB für DBpedia, vgl. Tabelle 3) einen zusätzlichen Overhead von mehreren GB für die Speicherung als String-Objekt erzeugen würden[7]. Der für das Kodieren erreichte Durchsatz für die LUBM-Datensätze ist vergleichbar mit den in [UMD+13] aufgeführten Ergebnissen (260 ktps), die zur Evaluation eines MapReduce basierten Verfahrens für das Dictionary-Encoding auf 32 Rechnern dienten. Das Dekodieren hingegen übersteigt mit rund 1.842 ktps die in [UMD+13] gezeigten Ergebnisse von 227 ktps um ein Vielfaches.

Tabelle 6 zeigt zusätzlich zu den bereits vorgestellten Performance Parametern, die durch das Dictionary-Encoding erreichte Komprimierung der Datensätze. Die komprimierten Dateien enthalten sowohl die Dictionary-Einträge als auch die für die Kodierung notwendige Hash-Struktur und Präfix-Summe. Auf diese Weise können Datensätze komprimiert gespeichert und zu einem späteren Zeitpunkt erneut verarbeitet werden, ohne dass die Möglichkeit der nachträglichen Kodierung weiterer Tripel oder Regel-Terme verloren geht.

7 Die Speicherung von Objekten in Java erzeugt einen zusätzlichen Overhead, der von verschiedenen Faktoren wie der Klassenzugehörigkeit und der verwendeten Java Virtual Machine abhängig ist.

Datensatz	Input (GB)		Output (GB)	Kompressions-rate
	plain	zip		
LUBM1000	11,3	0,35	1,22	9,3
LUBM2000	22,7	0,69	2,46	9,2
LUBM4000	45,5	1,38	4,92	9,2
LUBM8000	91,2	2,80	9,83	9,3
DBpedia	69,2	6,60	6,68	10,4
CTD	127,5	3,20	4,11	31,0

Tabelle 6: Ergebnisse der Komprimierung der verwendeten Datensätze durch das Dictionary-Encoding [PSZ15]

10.2.2 *Speicherverbrauch für das Reasoning*

Im vorherigen Kapitel wurde gezeigt, dass auch mit einer eingeschränkten Hauptspeicher-Verfügbarkeit das Dictionary-Encoding großer Datensätze möglich ist. Nach der Kodierung kann das gesamte Dictionary auf die Festplatte ausgelagert werden (vgl. Kapitel 9.1.2), so dass während des eigentlichen Reasonings-Vorgangs keine Belastung des Hauptspeichers durch die Datenstrukturen des Dictionary-Encoding stattfindet. Ebenso wurde in Kapitel 9.2 gezeigt, dass die Working-Memories des RETE-Algorithmus vollständig auf die Festplatte ausgelagert werden können. Lediglich die eingeführte Tripel-Index-Struktur zur Erkennung von Duplikaten muss im Hauptspeicher vorgehalten werden, um die hohe Frequenz der notwendigen Überprüfungen effizient bearbeiten zu können. Um einen Eindruck der notwendigen Überprüfungen vermitteln zu können, listet Tabelle 7 die Anzahl der Fakten für zwei exemplarische Datensätze auf, die beim Anwenden der RDFS bzw. ρdf Regeln abgeleitet werden und somit auf Duplikate überprüft werden müssen. Für den DBpedia-Datensatz beispielsweise werden während des Reasonings mittels der RDFS Regeln rund 2,3 Milliarden Tripel abgeleitet, die durch die Überprüfung auf Duplikate auf rund 82 Millionen reduziert werden.

Der notwendige Speicher für die Tripel-Index-Struktur hängt von verschiedenen Faktoren ab. Zum einen hat die Anzahl der im Datensatz enthaltenen Prädikate durch das Konzept des Vertical-Partitioning [AMMH07] einen direkten Einfluss auf die Anzahl der intern verwendeten Datenstrukturen. Für jedes Prädikat werden HashSets angelegt, die Werte mit einer Größe von 2 bis 16 Byte speichern können und damit der Repräsentation der kodierten Prädikat-Objekt-Kombination dienen. Zum anderen kann durch die Verwendung der HashSets ein Overhead entstehen, da entsprechend des gewählten Load-Faktors immer eine be-

Datensatz	Regelsatz	abgeleitete Tripel	eindeutige Tripel	Duplikate
LUBM2000	ρdf	529,9 M	66,7 M	463,2 M
LUBM2000	RDFS	1.813,6 M	110,1 M	1.703,5 M
DBpedia	ρdf	580,1 M	7,0 M	573.1 M
DBpedia	RDFS	2.263,6 M	81,5 M	2.182,1 M

Tabelle 7: Anzahl abgeleiteter Fakten [PSZ15]

stimmte Anzahl an freien Einträgen vorgehalten werden muss. Tabelle 8 zeigt den tatsächlichen Speicherverbrauch für die Tripel-Index-Struktur während der Anwendung der ρdf Regeln auf verschiedene Datensätze, der durch die Ausführung der Reasoner-Implementierung ermittelt wurde.

Datensatz	Anzahl Tripel	Anzahl der Prädikate	Byte pro Tripel	Byte pro Tripel mit Overhead	Speicherverbrauch gesamt
LUBM1000	167,0 k	32	6,04	9,71	1.620 MB
LUBM2000	333,7 k	32	6,11	9,29	3.098 MB
LUBM4000	667,5 k	32	5,69	8,26	5.513 MB
LUBM8000	1.335,1 k	32	6,16	8,80	11.748 MB
DBpedia	400,6 k	53.139	7,45	12.81	5.129 MB
CTD	335,2 k	43	5,94	9,36	3.137 MB

Tabelle 8: Speicherverbrauch für die Tripel-Index-Struktur nach Anwendung der ρdf Regeln [PSZ15]

Die in Tabelle 8 aufgeführten *Byte pro Tripel mit Overhead* zeigen den tatsächlichen durchschnittlichen Speicherverbrauch pro Tripel, der den Overhead durch die Hash-Strukturen berücksichtigt. Dieser ist ebenfalls in der Spalte *Speicherverbrauch gesamt* eingerechnet. Nicht berücksichtigt wurde hingegen ein möglicher Overhead, der durch die Erzeugung von Objekten (z.B. eines HashSets) in der Java-Umgebung entsteht. Der Grund hierfür ist die geringe Anzahl an erzeugten Objekten und der damit verbundene Overhead (je nach Art des Objektes rund 40 Byte), der im Gegensatz zu dem durch die eigentlichen Daten verbrauchten Speicher unwesentlich gering ist. Unter der Annahme, dass pro Prädikat in einem Datensatz beispielsweise rund 150 Byte für die Erzeugung von Objekten benötigt werden, beläuft sich der Overhead für den DBpedia-Datensatz mit rund 53.000 Prädikaten auf ca. 8 MB [PSZ15].

Die durchgeführten Versuche zeigen, dass der notwendige Speicher pro Tripel durch die Anwendung der verschiedenen Komprimierungsverfahren im Durchschnitt auf 6,2 Byte bzw. 9,7 Byte mit Overhead reduziert werden konnte. Demgegenüber steht ein Speicherverbrauch von 24 Byte für die einfache Repräsentation eines Tripels durch drei 64 Bit Datenwerte. Wird zusätzlich bei der Verwendung der einfachen Repräsentation ein Overhead im gleichen Verhältnis wie bei der komprimierten Speicherung zugrunde gelegt (dieser entsteht ebenfalls durch die Nutzung von Hash-Strukturen), werden für die Speicherung eines Tripels im Schnitt 37,5 Byte benötigt. Damit konnte durch die Nutzung der eingeführten Tripel-Index-Struktur eine Reduzierung des Speicherverbrauchs allein für die Hash-basierte Speicherung der Tripel um rund 74% erreicht werden, ohne dabei die Möglichkeit der effizienten Überprüfung auf Duplikate einbüßen zu müssen.

Der Gesamt-Speicherverbrauch des Reasoning-Prozesses reduziert sich hingegen durch die Maßnahmen des Dictionary-Encodings unter Verwendung des Festplattenspeichers, der Auslagerung der Working-Memories sowie der Verwendung der komprimierten Tripel-Index-Struktur z.B. für den LUBM2000 Datensatz bei der Anwendung der ρdf Regeln von rund 34,4 GB auf 3,1 GB. Somit wird für den betrachteten Datensatz durch die Anwendung der eingeführten Konzepte eine Reduzierung der Hauptspeicherbelastung um rund 90% erzielt.

10.2.3 *Effektivität der Parallelisierung*

In den vorherigen Kapiteln wurden verschiedene Möglichkeiten der Parallelisierung des RETE-Algorithmus vorgestellt. Neben der Node-Level-Parallelisierung [Gup86] ist durch die Einführung der Partitionierung der Arbeitslast in Kapitel 8 eine weitere Möglichkeit der Zerlegung der notwendigen Berechnungen in voneinander unabhängige Partitionen geschaffen worden (Host-Level-Parallelisierung). Diese können einerseits auf der CPU durch die Nutzung mehrerer Prozesse parallel verarbeitet werden und bieten im Gegensatz zu der in [Gup86] vorgeschlagenen Parallelisierung den Vorteil der gleichmäßigen Lastverteilung (eine dem Datensatz entsprechende Partitionsgröße vorausgesetzt). Andererseits lassen sich die Partitionen jeweils entsprechend der in Kapitel 7 vorgestellten Verarbeitung auf massiv paralleler Hardware ausführen.

Zur Evaluation der verschiedenen Konzepte wurde die bereits in [PBSZ13b] und [PBSZ14] eingeführte Reasoner-Implementierung erweitert und die zuvor skizzierte Anwendungs-Architektur implementiert (siehe Abbildung 25). Zusätzlich wurde die Möglichkeit der seriellen Ausführung aller Berechnungen sowie der parallelen Ausführung unter ausschließlicher Nutzung mehrerer Prozesse auf der CPU integriert. Somit ergeben sich drei Möglichkeiten der Ausführung des Reasoners,

die als *Serielle Strategie*, *Host-Level-Strategie* und *Device-Level-Strategie* referenziert werden:

SERIELLE STRATEGIE: Alle Berechnungen werden nacheinander durchgeführt, ohne eine parallele Verarbeitung oder eine Nutzung der GPU-Ressourcen zu berücksichtigen.

HOST-LEVEL-STRATEGIE: Es wird die in Abbildung 25 skizzierte Architektur mit mehreren Prozessen und einer parallelen Verarbeitung verwendet. Die Berechnung der einzelnen Arbeitslast-Partitionen wird nicht den GPU-Workern übergeben, sondern innerhalb der einzelnen Threads auf der CPU ausgeführt.

DEVICE-LEVEL-STRATEGIE: Es werden entsprechend der in Abbildung 25 skizzierten Architektur mehrere Threads zur Bereitstellung der Partitionen genutzt. Die Verarbeitung der Partitionen findet jeweils auf einer GPU statt.

Für die Durchführung der Versuche unter Verwendung der Host-Level-Strategie wurden G_x und G_y, also die Größe der Partitionen, für die unterschiedlichen Schritte des RETE-Algorithmus jeweils deutlich kleiner gewählt (100.000 x 20.480) als in Tabelle 4 angegeben, um die Anzahl der resultierenden Partitionen zu erhöhen und eine möglichst gleichmäßige Belastung der verwendeten Prozesse zu erzielen. Tabelle 9 zeigt eine Gegenüberstellung der drei verschiedenen Ausführungsstrategien für das pD* Reasoning auf unterschiedlichen Größen des Benchmark-Datensatzes, die auf der Workstation ausgeführt worden sind. Abbildung 26 visualisiert die Ergebnisse (unter Verwendung einer logarithmischen Skala), um den Unterschied der Ausführungsgeschwindigkeiten hervorzuheben.

Datensatz	Input-Tripel	Output-Tripel	Device-Level Strategie	Host-Level Strategie	Serielle Strategie
LUBM5	625 k	829 k	1,1 s	24,6 s	64,5 s
LUBM10	1.273 k	1.689 k	1,8 s	57,8 s	280,9 s
LUBM20	2.689 k	3.566 k	3,7 s	245,4 s	1221,0 s
LUBM40	5.309 K	7.038 k	10,2 s	874,0 s	4630,5 s

Tabelle 9: Ausführungszeit in Sekunden für das pD* Reasoning unter Verwendung der verschiedenen Ausführungsstrategien

Die Komplexität des pD* Regelsatzes drückt sich während der Ausführung des RETE-Algorithmus insbesondere in einer hohen Anzahl an notwendigen Operationen für das Beta-Matching aus, die bei einer Verdoppelung der Datensatzgröße

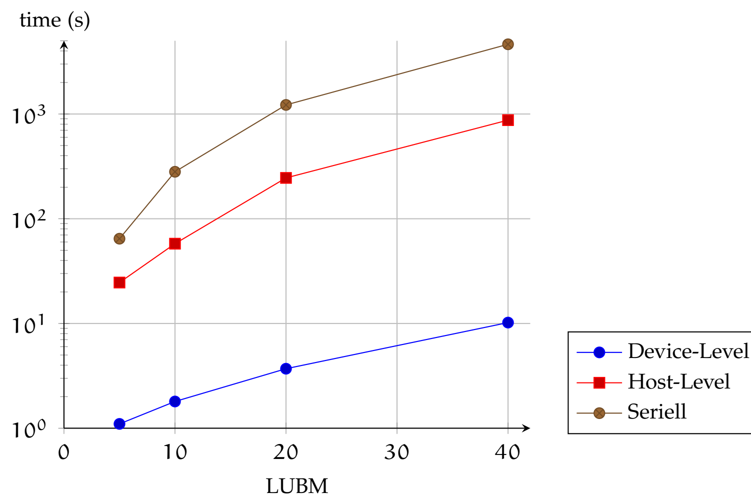


Abbildung 26: Vergleich des pD* Reasonings unter Verwendung der verschiedenen Ausführungsstrategien

zu einem exponentiell steigendem Aufwand führen, was insbesondere an der Ausführungszeit für die serielle Implementierung erkennbar ist. Während die Host-Level-Strategie bereits zu einer Performancesteigerung gegenüber der seriellen Ausführung mit einem Faktor von 5,3 für den LUBM40 Datensatz führt, kann durch die Verwendung der Device-Level-Strategie ein weiterer Geschwindigkeitszuwachs mit einem Faktor von etwa 86 gegenüber der Host-Level-Parallelisierung für selbigen Datensatz erzielt werden.

Im vorherigen Versuch wurden die in dieser Arbeit komplexesten Ableitungsregeln auf verhältnismäßig kleine Datensätze angewandt. Der folgende Versuch hingegen zeigt vor allem die aus den Konzepten der Parallelisierung resultierenden Unterschiede bei größeren Datensätzen und leichtgewichtigeren Ableitungsregeln. Dazu werden die RDFS Regeln auf Datensätze mit einer Größe von bis zu einer Milliarde Tripel angewandt. Als Ausführungsumgebung dient ebenfalls die Workstation. Die Ergebnisse sind in Tabelle 10 und Abbildung 27 dargestellt und zeigen, dass der Unterschied zwischen der Device- und Host-Level-Strategie im Gegensatz zu dem Versuch unter Verwendung der pD* Regeln mit einem durchschnittlichen Faktor von 2,7 deutlich geringer ausfällt, aber dennoch einen klaren Geschwindigkeitszuwachs zeigen.

Der Unterschied des Performance-Gewinns zwischen den betrachteten Regelsätzen kann auf die Anzahl der notwendigen Match-Operationen zurückgeführt werden. Allein für den LUBM40 Datensatz müssen für das Ableiten der pD* Regeln rund 400 Milliarden Beta-Match Operationen berechnet werden (eine Verdoppelung der LUBM-Datensatzgröße resultiert in einer Vervierfachung der not-

Datensatz	Input-Tripel	Output-Tripel	Device-Level Strategie	Host-Level Strategie	Serielle Strategie
LUBM1000	133.614 k	188.704 k	61 s	173 s	512 s
LUBM2000	267.028 k	377.121 k	121 s	342 s	1044 s
LUBM4000	534.204 k	754.788 k	307 s	725 s	2119 s
LUBM8000	1.068.395 k	1.508.890 k	720 s	1975 s	5006 s

Tabelle 10: Ausführungszeit in Sekunden für das RDFS Reasoning unter Verwendung der verschiedenen Parallelisierungskonzepte

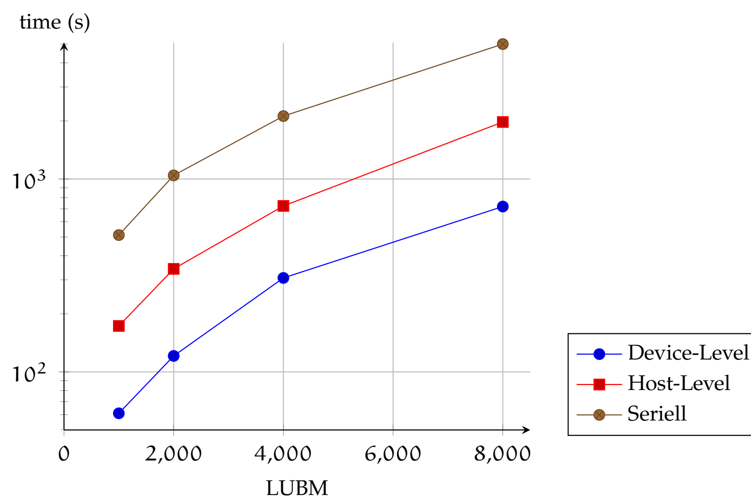


Abbildung 27: Vergleich des RDFS Reasonings unter Verwendung der verschiedenen Parallelisierungskonzepte

wendigen Beta-Match Operationen), die sich bei der Nutzung der RDFS Regel beim gleichen Datensatz auf rund eine Milliarden Operationen (0,25%) reduzieren (bei der Verwendung der RDFS Regeln führt eine Verdoppelung der LUBM-Datensätzgröße auch zu einer Verdoppelung der Beta-Match Operationen). Aufgrund der Effektivität des eingeführten Parallelisierungskonzepts unter Verwendung massiv paralleler Hardware kann die hohe Anzahl an Match-Operationen für die pD* Regeln besonders schnell verarbeitet werden, was zu einer Ausführungszeit von rund acht Sekunden für diesen Schritt führt. Die Berechnung auf der CPU hingegen nimmt ca. 869 Sekunden in Anspruch und führt so zum Geschwindigkeitsunterschied zwischen der Device- und Host-Level-Parallelisierung bzw. der seriellen Ausführung. Des Weiteren erklärt diese Beobachtung den deutlich geringeren Geschwindigkeitsvorteil, der beim Ableiten der RDFS Regeln im Verhältnis zu pD* durch die Verwendung der GPUs erreicht wird. Aufgrund der

wesentlich geringeren Anzahl (gemessen im Verhältnis zur Datensatzgröße) an notwendigen Match-Operationen für das Ableiten der RDFS Regeln kann die Ausführung auf der GPU die Gesamtausführungszeit nur bedingt beeinflussen.

Die durchgeführten Versuche haben gezeigt, dass die Konzepte der Parallelisierung des RETE-Algorithmus für die verwendete Konfiguration, bestehend aus Datensatz und Ableitungsregeln, zu einer deutlichen Performancesteigerung beitragen. Die Ausführungszeit der pD* Regeln auf dem LUBM40 Datensatz konnte z.B. unter Nutzung der massiv parallelen Hardware von zwei GPUs im Gegensatz zu einer seriellen Implementierung um über 99% reduziert werden. Für die Anwendung der RDFS Regeln wurde hingegen eine Reduzierung der Ausführungszeit um rund 87% erreicht. Insbesondere die Erkenntnis der wesentlich performanteren Berechnung der Match-Operationen auf der GPU zeigt, dass sich die parallele Architektur von Grafikkarten für die Implementierung eines effizienten, regelbasierten Reasoning-Prozesses nutzen lässt. Der tatsächlich erreichte Geschwindigkeitszuwachs kann jedoch in Abhängigkeit des verwendeten Regelsatzes und der Ausgangsdaten variieren, wie die Evaluation unter Verwendung der pD* und der RDFS Regeln gezeigt hat.

10.2.4 *Large-scale Reasoning auf einzelnen Rechnern*

Das primäre Ziel der vorliegenden Arbeit ist die Beantwortung der Frage, ob sich die massiv parallele Architektur moderner GPUs für die Entwicklung eines effizienten, regelbasierten Reasoners nutzen lässt. Die vorherigen Versuche haben bereits zur Beantwortung dieser Frage beigetragen und gezeigt, dass durch die eingeführten Konzepte unter Verwendung von GPUs eine zum Teil um ein Vielfaches gesteigerte Performance erreicht werden kann. Ein weiteres Ziel dieser Arbeit besteht darin, den notwendigen Ressourcenbedarf für den Reasoning-Prozess möglichst weit zu reduzieren, um auch Large-scale Reasoning mit bis zu mehreren Milliarden Tripeln auf einzelnen Rechnern ausführen zu können. Während z.B. das Dicitionary-Encoding sowie der Speicherverbrauch durch die Tripel-Index-Struktur bereits im Einzelnen betrachtet wurde, findet in diesem Kapitel eine Betrachtung des gesamten Reasoning-Prozesses und des erreichten Durchsatzes statt, der anschließend für den Vergleich mit verwandten Arbeiten genutzt wird. Dazu werden einerseits DBpedia und CTD als Datensätze verwendet, aber auch der Benchmark-Datensatz wird in verschiedenen Skalierungen genutzt, um das Verhalten bei Datenmengen von bis zu 3,2 Milliarden Eingabe-Fakten zu evaluieren.

Tabelle 11 fasst die Ergebnisse für das ρdf und RDFS Reasoning auf dem MacBook zusammen. Während der Durchsatz für die Benchmark-Datensätze mit steigender Größe sowohl für den ρdf- als auch für den RDFS-Regelsatz abnimmt,

Datensatz	Input Tripel	Gesamt-Tripel ρdf	Reasoning ρdf	Durchsatz ρdf	Gesamt-Tripel RDFS	Reasoning RDFS	Durchsatz RDFS
LUBM1000	134 M	167 M	41,4 s	4,02 M	189 M	114 s	1,65 M
LUBM2000	267 M	334 M	98,4 s	3,39 M	377 M	288 s	1,31 M
LUBM4000	534 M	668 M	296,9 s	2,25 M	754 M	758 s	1,00 M
LUBM8000	1.068 M	1.335 M	716,7 s	1,86 M	1.509 M	1.825 s	0,83 M
DBpedia	394 M	401 M	409,9 s	1,16 M	475 M	2.887 s	0,165 M
CTD	335 M	358 M	70,2 s	5,10 M	358 M	306,8 s	1,18 M

Tabelle 11: Ausführungszeiten für das ρdf und RDFS Reasoning auf dem MacBook [PSZ15]

konnte dennoch gezeigt werden, dass das Reasoning mit über einer Milliarde Tripel (LUBM8000) durch die eingeführten Konzepte auf einem Laptop möglich ist. Dabei wurde ein maximaler Durchsatz von rund 4 Millionen Tripel pro Sekunde (Mtps) für den Benchmark-Datensatz und die ρdf Regeln erzielt. Für das RDFS Reasoning liegt dieser hingegen bei rund 1,7 Mtps. Weiterhin zeigen die Ergebnisse, dass die eingeführten Konzepte nicht nur für die künstlichen Benchmark-Datensätze zielführend sind, sondern auch für die mehrere hundert Millionen Tripel großen Datensätze DBpedia und CTD, für die z.B. für die ρdf Regeln ein Durchsatz von rund 1,2 Mtps bzw. 5,1 Mtps erreicht wurde.

Im Gegensatz zum MacBook ist die Workstation mit zwei leistungsstarken Grafikkarten und deutlich mehr Speicher ausgestattet. Entsprechend wurde für die Durchführung der Evaluation die Arbeitslast erhöht, indem Datensätze mit bis zu 3,2 Milliarden Tripel genutzt wurden. Ähnlich wie beim MacBook sinkt auch auf der Workstation mit einer wachsenden Datensatzgröße der Durchsatz, der für den Benchmark-Datensatz zwischen 6,7 Mtps und 2,2 Mtps bei der Anwendung der ρdf Regeln liegt (vgl. Tabelle 12) und damit fast doppelt so hoch ist wie auf dem MacBook. Einen noch deutlicheren Anstieg des Durchsatzes konnte für DBpedia (rund 12 Mtps) und CTD (knapp 19 Mtps) erzielt werden, der nicht nur aus der Verteilung der Arbeitslast auf mehrere GPUs resultiert, sondern im Besonderen auch während des Beta-Matchings von der Leistung der einzelnen GPUs profitiert. Anknüpfend an die Erkenntnis aus der vorherigen Evaluation, die bereits die hohe Effektivität der Berechnung von Beta-Match Operationen unter Verwendung massiv paralleler Hardware gezeigt hat, führt die Verwendung der leistungsstärkeren GPUs der Workstation im Gegensatz zu der Laptop-GPU zu einem zusätzlichem Geschwindigkeitsvorteil. Dieser kann an der Verteilung der Gesamtausführungszeit auf die drei wesentlichen Schritte des Reasonings mittels des RETE-Algorithmus, das Alpha-Matching, Beta-Matching und das Ableiten

Datensatz	Input Tripel	Gesamt-Tripel ρdf	Reasoning ρdf	Durchsatz ρdf	Gesamt-Tripel RDFS	Reasoning RDFS	Durchsatz RDFS
LUBM1000	134 M	167 M	25,7 s	6,50 M	189 M	61,3 s	3,08 M
LUBM2000	267 M	334 M	49,9 s	6,69 M	377 M	121,3 s	3,11 M
LUBM4000	534 M	668 M	113,0 s	5,91 M	754 M	306,9 s	2,46 M
LUBM8000	1.068 M	1.335 M	289,2 s	4,62 M	1.509 M	720,0 s	2,10 M
LUBM16000	2.136 M	2.669 M	1000 s	2.67 M	3.017 M	2.517 s	1.20 M
LUBM24000	3.204 M	4.004 M	1.855 s	2,18 M	4.525 M	4.428,0 s	1,02 M
DBpedia	394 M	401 M	33,2 s	12,07 M	475 M	260,9 s	1,82 M
CTD	335 M	358 M	19,1 s	18,80 M	358 M	84,5 s	4,24 M

Tabelle 12: Ausführungszeiten für das ρdf und RDFS Reasoning auf der Workstation

von Fakten verdeutlicht werden, deren zeitliches Verhältnis für das Anwenden der ρdf Regeln auf dem DBpedia-Datensatz in Abbildung 28 dargestellt ist.

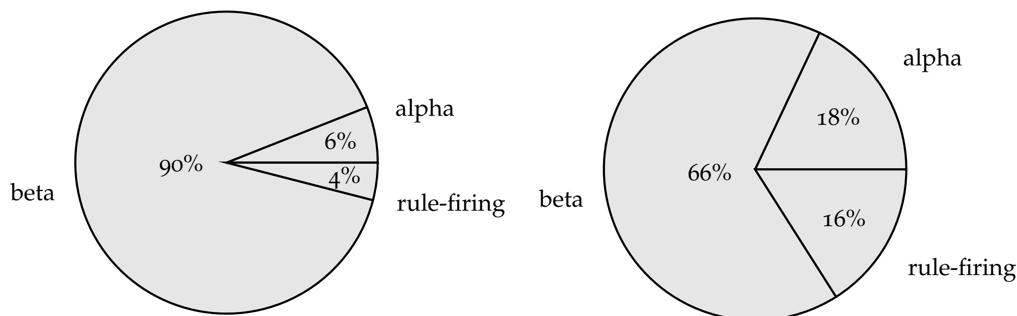


Abbildung 28: Darstellung der Reasoning-Zeit verteilt auf das Alpha-Matching, Beta-Matching und Feuern von Regeln für DBpedia und die ρdf Regeln auf dem MacBook (links) und der Workstation (rechts)

Bei der Verwendung der Workstation als Ausführungsumgebung reduziert sich im Vergleich zum MacBook nicht nur die Laufzeit des Reasonings, sondern auch der vom Beta-Maching ausgehende Anteil an der Ausführungszeit. Das Verhältnis zwischen der Laufzeit für das Alpha-Matching und das Ableiten von Fakten ist hingegen bei beiden Ausführungsumgebungen nahezu konstant.

Eine Erklärung für die Verschiebung der Anteile liefert auch Abbildung 29, die die jeweilige GPU-Nutzung während der einzelnen Schritte auf dem MacBook und der Workstation aufschlüsselt. Während des Alpha-Matchings wird die GPU auf dem MacBook beispielsweise nur zu 38% der Ausführungszeit verwendet. Diese Zeit beinhaltet bereits den Transfer von Daten zwischen der GPU und dem Host-System, so dass 62% der Ausführungszeit des Alpha-Matchings für die Bereitstel-

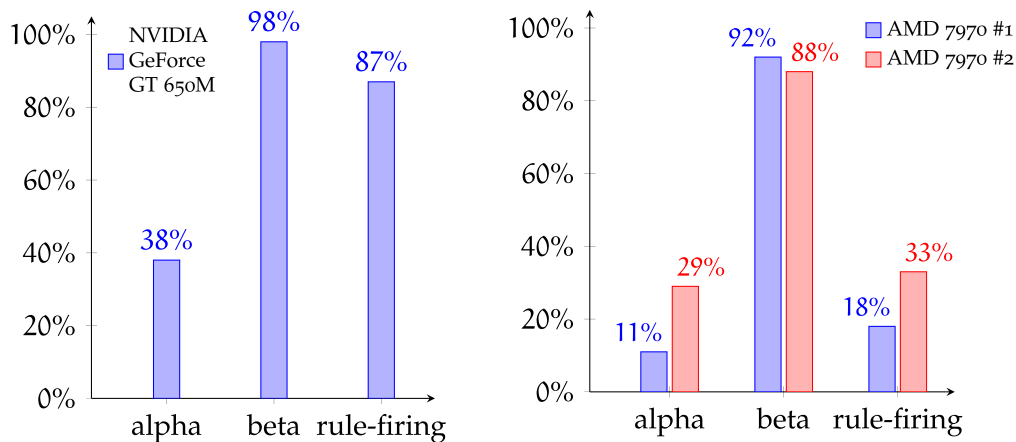


Abbildung 29: Auslastung der GPU(s) während der einzelnen Reasoning-Schritte auf dem MacBook (links) und auf der Workstation (rechts) für die Anwendung der ρdf Regeln auf DBpedia

lung, Transformation und Speicherung der Daten beansprucht werden. Während des Beta-Matchings hingegen wird eine nahezu vollständige Auslastung erreicht, so dass die Ausführungszeit für das Beta-Matching direkt von der Laufzeit der Berechnung auf der GPU abhängig ist. Die zeitliche Beanspruchung der GPUs der Workstation ist während des Beta-Matchings etwas niedriger, was zusammen mit der Erkenntnis der ungleichmäßigen Auslastung der beiden Grafikkarten ebenso wie beim Feuern der Regeln auf eine Begrenzung der Ausführungsgeschwindigkeit durch die Host-seitige Implementierung der Anwendung deutet. Somit stellt für den betrachteten Datensatz die Grafikkarte des MacBooks den limitierenden Faktor innerhalb der Ausführungsumgebung dar, während dieser bei der Workstation aufgrund der wesentlich höheren Leistung der GPUs auf die Host-seitige Implementierung zurückzuführen ist. Entsprechend könnte eine noch effizientere Realisierung der Host-seitigen Anwendung, die z.B. die Bereitstellung und Speicherung sämtlicher Daten beinhaltet und demnach auch abhängig von der Lese- und Schreibgeschwindigkeit der verwendeten Hardware ist, insbesondere auf der Workstation zu einer zusätzlichen Performancesteigerung für das Reasoning mittels leichtgewichtiger Ontologiesprachen beitragen.

10.3 VERGLEICH ZU VERWANDTEN ARBEITEN

Ein Überblick über verwandte Arbeiten und eine Kategorisierung des parallelen Reasonings in das *verteilte Reasoning*, zu dem beispielsweise auch die MapReduce-basierten Ansätze zählen, sowie das *Reasoning unter Verwendung (massiv) paralleler Hardware* wurde bereits in Kapitel 3.3.2 gegeben. Nicht alle der in den vorherigen

Kapiteln genannten Arbeiten können z.B. aufgrund ihrer Ausrichtung auf eine spezielle Semantik oder einen besonderen Anwendungsfall direkt zu einem Vergleich herangezogen werden. Andere Arbeiten hingegen erlauben beispielsweise durch den erzielten Durchsatz für den LUBM-Benchmark-Datensatz einen Vergleich und werden in diesem Kapitel erneut aufgegriffen.

Zu den Arbeiten, die sich auf die Verarbeitung semantischer Daten auf einzelnen Rechnern beschränken, gehörten z.B. [WED+08], [UMJ+13] und [HP12]. In [WED+08] wurde ein Ansatz vorgestellt, der neben dem RDFS Reasoning auch benutzerdefinierte Regeln unterstützt und basierend auf einer Oracle Datenbank implementiert wurde. Für die Evaluation auf insgesamt drei Rechnern, von denen die Datenbankanwendung auf einem ausgeführt wurde und die anderen beiden der Bereitstellung von zusätzlichem Festplattenspeicher dienten, wurde ebenfalls der Lehigh University Benchmark mit einer maximalen Größe von rund 133 Millionen Tripeln (LUBM1000) genutzt. Die Ausführungszeit für das RDFS Reasoning auf diesem Datensatz betrug 6 Stunden und 34 Minuten. Mit den in dieser Arbeit vorgestellten Ansätzen konnte hingegen für den gleichen Datensatz auf dem MacBook eine Ausführungszeit von rund 114 Sekunden erzielt werden, während die Ausführungszeit auf der Workstation bei ca. 61 Sekunden liegt.

In [UMJ+13] stellen Urbani et al. eine Arbeit vor, die nicht nur ein Reasoning auf statischen Informationen ermöglicht, sondern auch auf RDF-Streams. Während damit die originäre Arbeit einen anderen Fokus besitzt, lässt sich dennoch ein Vergleich in Bezug auf die vollständige Materialisierung des in [UMJ+13] genutzten RDFS Fragments ρdf anstellen. Für den LUBM8000 Datensatz wurde in [UMJ+13] auf einem Rechner mit einem Quad-Core Prozessor und 24 GB Speicher ein Durchsatz von 227 ktps erreicht. Der Durchsatz für den in dieser Arbeit vorgestellten Ansatz hingegen liegt bei dem LUBM8000 Datensatz bei etwa 1,9 Mtps auf dem MacBook und 4,6 Mtps auf der Workstation und übersteigt damit den in [UMJ+13] erreichten um ein Vielfaches.

Eine Arbeit, die ebenfalls die parallele Architektur von modernen CPUs und GPUs nutzt, wurde in [HP12] von Heino et al. vorgestellt (im Folgenden als *grdfs-Reasoner* bezeichnet) und bereits in [PBSZ13b] für eine Gegenüberstellung mit den grundlegenden Konzepten dieser Arbeit genutzt. Im Gegensatz zur vorliegenden Arbeit wird in [HP12] ausschließlich die ρdf Semantik implementiert, ohne benutzerspezifische Regeln zu unterstützten. Des Weiteren ist der vorgestellte Ansatz limitiert auf die Verwendung einer einzelnen Grafikkarte, so dass keine Lastverteilung oder Skalierung durch Hinzunahme weiterer Hardware möglich ist. Tabelle 13 zeigt die Ergebnisse des in [HP12] vorgestellten Reasoners im Vergleich zu der in dieser Arbeit vorgestellten Implementierung. Als Ausführungsumgebung für die durchgeführte Gegenüberstellung wurde für beide Reasoner die zuvor beschriebene Workstation genutzt. Um einen besseren Vergleich aufstellen zu kön-

Datensatz	Input Tripel	Output Tripel	Ausführungszeit grdfs Reasoner [HP12]	Ausführungszeit der eigenen Implementierung	Faktor
LUBM125	16.7 M	20,9 M	12,48 s	4,60 s	2,71
LUBM250	33,4 M	41,7 M	24,94 s	8,35 s	2,99
LUBM500	66.8 M	83,4 M	50,89 s	15,89 s	3,20
LUBM1000	133,6 M	167,0 M	191,76 s	30,17 s	6,36

Tabelle 13: Vergleich der Ausführungszeiten der in [HP12] vorgestellten Arbeit und der in dieser Arbeit eingeführten Implementierung unter Verwendung einer einzelnen AMD 7970 GPU

nen, wird bei der Ausführung der in dieser Arbeit vorgestellten Implementierung auf die Verwendung der zweiten GPU verzichtet.

Die Ergebnisse zeigen, dass der in dieser Arbeit vorgestellte Ansatz trotz der Spezialisierung des grdfs-Reasoners auf die ρdf Semantik mindestens um den Faktor 2,7 schneller ist. Des Weiteren konnte festgestellt werden, dass beim LUBM1000 Datensatz die Ergebnisse des grdfs-Reasoners nicht mehr deterministisch sind, was ebenso wie die stark angestiegene Laufzeit auf eine Überschreitung des verfügbaren Speicherplatzes auf der verwendeten GPU (3 GB) schließen lässt. Somit ist der in dieser Arbeit eingeführte Ansatz nicht nur schneller, sondern auch in der Lage wesentlich größere Datensätze zu verarbeiten.

In [MNP+14a] und [MNP+14b] stellen Motik et al. RDFox vor, eine Reasoner-Implementierung für das Anwenden von Datalog-Programmen unter Verwendung von Multicore-Prozessoren, die mittels der OWL 2 RL Semantik evaluiert wird. Ebenso wie in dieser Arbeit wird von Motik et al. eine Hash-basierte Index-Struktur aufgebaut, die jedoch nicht ausschließlich der Erkennung von Duplikaten dient, sondern auch für die Navigation durch die Tripel während des Reasoning-Vorgangs genutzt wird. Entsprechend enthält die Struktur zusätzliche Datenelemente, so dass pro Tripel in der in [MNP+14b] gezeigten Evaluation für die im Hauptspeicher gehaltenen Datenstrukturen mindestens 80 Byte pro Tripel benötigt werden. Die Evaluation erfolgt auf einem Rechner mit 128 GB Arbeitsspeicher und wird mit Datensätzen durchgeführt, die nach Anwenden der Semantik bis zu 1,66 Milliarden Tripel beinhalten. Unter Verwendung von 32 Prozessorkernen (die Anzahl der verwendeten Prozessorkerne hat einen direkten Einfluss auf den Speicherverbrauch, so dass z.B. der Datensatz mit 1,66 Milliarden Tripel nur mit maximal acht Kernen verarbeitet werden konnte) konnte die Ausführungsgeschwindigkeit im Vergleich zu einer seriellen Implementierung maximal um den Faktor 19,5 gesteigert werden [MNP+14b].

Das Reasoning des RDFS subsets ρdf wird in [GM10] und [GJM+11] auf einem Supercomputer mit insgesamt 512 Prozessoren ausgeführt. Der parallele Algorith-

mus, der ausschließlich auf die Ausführung der ρdf Regeln ausgelegt ist, nutzt lediglich den Hauptspeicher und wird in [GJM+11] mit LUBM Datensätzen evaluiert, die nach dem Reasoning-Vorgang eine Größe von bis zu 20 Milliarden Tripel aufweisen. Diese Datensatzgröße wird in [GM10] gleichzeitig als Maximalbelastung der verwendeten Hardware mit insgesamt 4 TB Speicher angegeben, was auf einen Hauptspeicherverbrauch von rund 200 Byte pro Tripel schließen lässt. Bei der durchgeführten Evaluation wird unter Verwendung von 512 Prozessoren ein Durchsatz von bis zu 13,7 Mtps erreicht.

Neben den verwandten Arbeiten, die sich auf die Ressourcen eines einzelnen Rechners beschränkten bzw. einen Supercomputer verwenden, existieren verschiedene Ansätze, die die Arbeitslast z.B. unter Verwendung des MapReduce [DG04] Programmiermodells auf mehrere Rechner verteilen. Das Reasoning mittels der ρdf und pD* Regeln auf Datensätzen von bis zu 100 Milliarden Tripeln wurde von Urbani et al. in [UKM+10] und [UKM+12] gezeigt. Die Evaluation des MapReduce basierten WebPie-Reasoners erfolgt auf einem Cluster mit bis zu 64 Rechnern und erreicht für die ρdf Regeln einen Durchsatz von maximal 2,1 Mtps auf einem LUBM Datensatz mit insgesamt rund 20 Milliarden Tripeln. Für den LUBM8000 Datensatz wird hingegen ein Durchsatz von 481 ktps erzielt. Im Gegensatz zu den Konzepten aus dieser Arbeit resultiert die Verteilung der Arbeitslast auf mehrere Rechner zunächst in einem zusätzlichem Overhead, der in [UKM+12] auch für den niedrigeren Durchsatz bei kleineren Datensätzen angegeben wird. Während Urbani et al. gezeigt haben, dass WebPie ein skalierbares Reasoning erlaubt, ist der erreichte Durchsatz auf 64 Rechnern beispielsweise für den LUBM8000 Datensatz um den Faktor 9,6 langsamer als das Reasoning mit den in dieser Arbeit eingeführten Konzepten auf einem einzigen Rechner, der mit zwei GPUs ausgestattet ist.

In [WH09] führen Weaver und Hendler eine Strategie für das parallele RDFS Reasoning basierend auf einem Ansatz der Workload-Partitionierung ein, dessen Arbeitslast ebenfalls auf einem Rechencluster verteilt wird. Die für die Evaluation verwendeten Datensätze wurden aus einem LUBM10k Datensatz abgeleitet (LUBM10000) und auf verschiedene Größen skaliert, indem z.B. nur jedes vierte Instanz-Tripel verwendet wird. Der größte Datensatz (LUBM10k/4) mit rund 345,5 Millionen Tripel konnte durch 128 Prozesse in rund 291 Sekunden verarbeitet werden (entspricht einem Durchsatz von knapp 1,2 Mtps), ohne jedoch eine Erkennung bzw. Vermeidung von doppelt abgeleiteten Tripeln durchzuführen.

10.4 ERGEBNISAUSWERTUNG

Die durchgeführte Evaluation zeigt, dass die in dieser Arbeit eingeführten Konzepte geeignet sind, um den Hardware-Einsatz für das Large-scale Reasoning

mit leichtgewichtigen Ontologiesprachen drastisch zu reduzieren. Die Vorverarbeitung von Datensätzen durch das Dictionary-Encoding ist der erste Schritt, der für den Reasoning-Prozess durchgeführt werden muss. Dieser wurde für Datensätze mit bis zu einer Milliarde Tripel auf einem Laptop bzw. mit bis zu 3,2 Milliarden Tripeln auf einer Workstation ausgeführt, was durch eine zusätzliche Nutzung des Festplattenspeichers während des Kodier-Vorgangs ermöglicht wird. Gegenüber einer Verteilung der Arbeitslast während des Dictionary-Encodings auf mehrere Rechner (z.B. durch einen MapReduce basierten Ansatz mit 32 Rechnern [UMD+13]) ergibt sich durch die eingeführten Konzepte der Vorteil der zentralen Verarbeitung, wodurch ein zusätzlicher Overhead durch die Kommunikation zwischen verschiedenen (entfernten) Prozessen entfällt und trotz der beschränkten Ressourcen ein zu [UMD+13] vergleichbarer Durchsatz erzielt werden kann. Die Größe der verarbeitbaren Daten ist jedoch beschränkt durch den verfügbaren Arbeitsspeicher und kann nicht beliebig skaliert werden. Somit eignet sich das eingeführte Konzept für die in dieser Arbeit definierte Zielsetzung der Verarbeitung von Large-scale-Datensätzen mit bis zu mehreren Milliarden Tripeln auf einem einzelnen Rechner. Für die Verarbeitung beliebig großer Datensätze hingegen sind weitere Maßnahmen notwendig, die sich z.B. an der Arbeit in [UMD+13] orientieren können und eine Möglichkeit der Hardware-Skalierung bieten müssen.

Der Einfluss der Parallelisierung durch die Verwendung massiv paralleler Hardware in Bezug auf die Ausführungszeit des Reasonings ist direkt abhängig von der gewählten Semantik sowie dem gewählten Datensatz. In Kapitel 10.2.3 wurde dieser Unterschied durch die Verwendung der pD* Regeln im Vergleich zu den RDFS Regeln deutlich, die jeweils unter Nutzung der Device-Level, Host-Level und der seriellen Strategie angewandt wurden. Insbesondere für die pD* Semantik wird durch die Nutzung von zwei GPUs gegenüber einer seriellen Implementierung eine bis zu 450-fach gesteigerte Ausführungsgeschwindigkeit erreicht. Für die RDFS Regeln hingegen beschränkt sich der Geschwindigkeitszuwachs auf den Faktor acht. Der Unterschied resultiert aus der jeweiligen Anzahl der notwendigen Beta-Match Operationen, deren Anzahl im Verhältnis zur Datensatzgröße für das pD* Reasoning wesentlich größer ist als für das Ableiten der RDFS Regeln. Gleichzeitig kann diese Operation durch das eingeführte Konzept der Deviceseitigen Parallelisierung deutlich effizienter ausgeführt werden als unter alleiniger Nutzung der CPU, was zu der unterschiedlichen Performancesteigerung für die verschiedenen Regelsätze führt. Diese Ergebnisse zeigen, dass die massiv parallele Hardware sehr wirkungsvoll für die Gestaltung des Matching-Prozesses des RETE-Algorithmus genutzt werden kann. Somit kann der Einsatz moderner GPUs für die Implementierung eines regelbasierten Reasoners zu einer deutlich höheren Ausführungsgeschwindigkeit beitragen.

Neben der Betrachtung der Effektivität der Parallelisierungskonzepte leitet sich ein zweiter Schwerpunkt der durchgeführten Evaluation aus der Forschungsfrage der Reduzierung des Ressourcenbedarfs für den Reasoning-Prozess ab. Zu Beginn der Evaluation wurde eine Kalkulation des Speicherbedarfs aufgestellt, der von einer Implementierung eines Reasoners basierend auf dem RETE-Algorithmus in Anspruch genommen wird. Durch das RDFS Reasoning auf dem DBpedia-Datensatz (knapp 400 Millionen Tripel) ergibt sich ein Verbrauch von rund 42 GB Arbeitsspeicher, während der LUBM2000 Datensatz (knapp 270 Millionen Tripel) über 34 GB Speicher in Anspruch nimmt. Die in dieser Arbeit eingeführten Konzepte tragen dazu bei, dass trotz des originär hohen Speicherbedarfs nicht nur die beiden zuvor genannten Datensätze auf einem Laptop verarbeitet werden können, sondern auch ein Datensatz mit über einer Milliarde Tripel, für dessen Verarbeitung zuvor ein Rechner mit 192 GB Hauptspeicher notwendig war [PBSZ14]. Die Reduzierung des Speicherbedarfs um rund 90% wird einerseits durch Konzepte erreicht, die eine vollständige Auslagerung der Dictionary-Informationen sowie der Working-Memories auf die Festplatte erlauben. Andererseits reduziert die eingeführte Tripel-Index-Struktur durch verschiedene Verfahren der Datenkomprimierung den Speicherbedarf, der für den Vorgang der Erkennung von Duplikaten mittels einer Hauptspeicher-basierten Datenstruktur notwendig ist.

Ein Vergleich mit verwandten Arbeiten zeigt, dass die Ergebnisse dieser Arbeit in Bezug auf den Ressourcenverbrauch, aber auch in Bezug auf die Ausführungsgeschwindigkeit für das betrachtete Reasoning mit leichtgewichtigen Ontologiesprachen den aktuellen State of the Art übertreffen. Ein noch höherer Durchsatz für das Reasoning mit den LUBM Datensätzen wurde bisher nur unter Verwendung eines Supercomputers mit 512 Prozessoren und 4 TB Speicher veröffentlicht. Die Möglichkeit des Reasonings über einen Datensatz mit 1 Milliarde Tripel auf einem Laptop ist hingegen nicht bekannt. Somit tragen die eingeführten Konzepte nicht nur dazu bei, eine schnellere Verarbeitung semantischer Informationen zu erzielen, sondern auch den dazu notwendigen Aufwand und Ressourceneinsatz zu reduzieren. Dennoch ersetzt der in dieser Arbeit vorgestellte Ansatz nicht die bereits existierenden Verfahren, sondern stellt eine weitere Alternative dar, die in Abhängigkeit von der verfügbaren Hardware, der verwendeten Datensätze sowie der anzuwendenden Semantik gewählt werden kann. Insbesondere bei der Verwendung komplexerer Semantik oder von Datensätzen, deren Größe die Möglichkeiten eines einzelnen Rechners übersteigen, kann z.B. der Einsatz von MapReduce basierten Ansätzen aufgrund ihrer Skalierbarkeit bzw. die gezielte Implementierung einer speziellen Semantik zielführender sein.

Teil III

ZUSAMMENFASSUNG UND AUSBLICK

11 ZUSAMMENFASSUNG

Die vorliegende Arbeit greift die Fragestellung auf, ob sich die (massiv) parallele Hardware moderner Mehrkernprozessoren für einen effizienten Reasoning-Prozess unter Verwendung von Regeln auf semantischen Daten nutzen lässt. Als Reasoning wird allgemein das Materialisieren von implizit gegebenen Fakten aus einer explizit gegebenen Datenbasis bezeichnet [Fur14], das auf einer formalen Semantik basiert und beispielsweise bei Anfragen an eine Wissensdatenbank die Ergebnismenge durch die implizit enthaltenen Antworten erweitert. Vor dem Hintergrund der stetig wachsenden Menge an verfügbaren semantischen Informationen, die sich sowohl in der Größe einzelner, häufig domänenspezifischer Datensätze als auch in der Betrachtung der Gesamtmenge an verfügbaren Informationen, beispielsweise in der LOD-Cloud [SBP14], ausdrückt, müssen Konzepte für das Reasoning mit entsprechenden Large-scale-Datensätzen (als Large-scale werden in dieser Arbeit Datensätze mit mehreren hundert Millionen bis wenige Milliarden Fakten verstanden) gefunden werden. Während bereits eine Vielzahl an Arbeiten existiert, die Large-scale Reasoning unter Verwendung einer speziellen Semantik, basierend auf einer parallelen Verarbeitung auf einer Cluster-Infrastruktur, erlauben, wird die bereits aufgegriffene Fragestellung in dieser Arbeit um die Betrachtung des Reasonings unter beschränkten Ressourcen sowie der Definition der anzuwendenden Semantik über Regeln erweitert. Aus dieser Fragestellung leitet sich die Zielsetzung der Erforschung einer Reasoner-Architektur ab, die durch die Verwendung von massiv paralleler Hardware Large-scale Reasoning für leichtgewichtige Ontologiesprachen auf einzelnen Rechnern erlaubt. Die Infrastruktur soll zudem nicht abhängig von einer speziellen Semantik sein, sondern die Definition der Ableitungsvorschriften über Regeln ermöglichen.

Als Ausgangspunkt für die in dieser Arbeit eingeführten Reasoner-Architektur dient der von Forgy eingeführte RETE-Algorithmus [For82] [For79], der ein weit verbreiteter Pattern-Matching-Algorithmus zur Implementierung von Production Systems ist und sich zur Implementierung eines regelbasierten Reasoners eignet. Die Betrachtung der existierenden Optimierungs- und Parallelisierungskonzepte zeigt, dass viele Arbeiten auf der Struktur des für den RETE-Algorithmus aus den gegebenen Regeln abgeleiteten Netzwerks aufbauen. Der Grad einer möglichen Parallelisierung ist bei dieser Vorgehensweise direkt abhängig von der Anzahl der im Netzwerk vorhandenen Knoten (und damit von den verwendeten Regeln), während die Ausgewogenheit der Lastverteilung zusätzlich durch die Eingabedaten beeinflusst wird. Eine alternative Vorgehensweise zur parallelen Verarbeitung

stellt die Partitionierung der zu verarbeitenden Daten dar, die einerseits die Herausforderung einer gleichmäßigen Lastverteilung mit sich bringt und andererseits einen zusätzlichen Kommunikationsaufwand zwischen Prozessen für den Austausch neu abgeleiteter Fakten bedingt.

Die Betrachtung des Ausführungs- und Programmiermodells von OpenCL zur Programmierung moderner Grafikprozessoren zeigt, dass für eine effiziente Ausführung auf massiv paralleler Hardware eine Zerlegung des zu bearbeitenden Problems in eine Vielzahl voneinander unabhängiger Berechnungen notwendig ist. Um diese möglichst hohe Parallelisierung zu erreichen, wird für den in dieser Arbeit vorgestellten Ansatz neben einer Node-Level-Parallelisierung [Gup86] eine Zerlegung der Arbeitslast basierend auf einzelnen Tripeln bzw. einzelnen Matches des RETE-Algorithmus eingeführt. Der Grad der so erreichten Parallelisierung entspricht in Abhängigkeit des Ausführungsschrittes des Algorithmus der Anzahl der zu verarbeitenden Tripel bzw. der Anzahl der zu verarbeitenden Matches, während die jeweilige Operation nur verhältnismäßig wenig Rechenleistung beansprucht. Gleichzeitig wird durch die Parallelisierung basierend auf der Ebene einzelner Match-Operationen statt z.B. der Nutzung eines Ansatzes der Daten-Partitionierung, ein zusätzlicher Kommunikationsaufwand zwischen Prozessen vermieden und eine gleichmäßige Lastverteilung ermöglicht. Aufbauend auf dieser Grundlage trägt ein Konzept der Partitionierung der Match-Operationen dazu bei, dass einerseits eine Verteilung der Arbeitslast auf mehrere Grafikprozessoren möglich ist und andererseits die Größe der Eingabedaten nicht durch den Speicher der verwendeten GPU(s) beschränkt wird. Des Weiteren erlaubt diese Methode neben der bereits erwähnten Node-Level-Parallelisierung eine zusätzliche parallele Verarbeitung durch mehrere Prozesse auf der CPU, so dass die Ressourcen der in einem System vorhandenen CPUs und GPUs genutzt werden können.

Neben der Einführung eines Konzepts zur Übertragung des RETE-Algorithmus auf eine hoch parallele Ausführungsumgebung betrachtet die vorliegende Arbeit den Ressourcenverbrauch für den Reasoning-Prozess, der häufig maßgeblich für die Beschränkung der Datensatzgröße für die Verarbeitung auf einem System ist. Beispielsweise konnte festgestellt werden, dass bereits für das RETE-basierte RDFS Reasoning auf einem Datensatz mit rund 380 Millionen Tripel über 34 GB Arbeitsspeicher benötigt werden, wovon die Hälfte des Speicherbedarfs auf die Strukturen des RETE-Algorithmus zurückzuführen ist. Ein weiterer großer Teil des verwendeten Speicherplatzes wird für die Bereitstellung der Tripel in einer Dictionary-kodierten Form benötigt. Zur Adressierung der Herausforderung des hohen Speicherverbrauchs werden verschiedene Konzepte umgesetzt. Während des vorverarbeitenden Schrittes des Dictionary-Encoding wird ein Verfahren angewandt, dass zusätzlich Festplattenspeicher verwendet und nach Abschluss der Datensatzkodierung den gesamten Hauptspeicher wieder freigibt. Der darauf auf-

bauende Reasoning-Prozess ist so gestaltet, dass sowohl die Tripel als auch die Zwischenergebnisse des RETE-Algorithmus auf der Festplatte gespeichert werden können. Um diese Auslagerung ohne eine massive Beeinflussung der Ausführungszeit durch die erhöhten Zugriffszeiten zu ermöglichen, werden alle Informationen so abgelegt, dass sie in zusammenhängenden Blöcken gelesen und geschrieben werden können. Der daraus resultierenden komplexeren Ausführung der Algorithmik auf den Grafikprozessoren wird mit einem generativen Ansatz begegnet, der zu Beginn des Reasoning-Prozesses sämtlichen Quellcode zur Ausführung auf der GPU, basierend auf den gegebenen Regeln, generiert. Diese Vorgehensweise ermöglicht die Reduzierung von Übergabeparametern, Kontrollstrukturen und Speicherzugriffen und trägt somit zu einer effizienteren Ausführung der Berechnungen auf massiv paralleler Hardware bei.

Der für den Reasoning-Prozess benötigte Speicher reduziert sich nach Anwenden der zuvor genannten Konzepte auf den Speicherbedarf einer Datenstruktur, die der Erkennung von mehrfach abgeleiteten Fakten (Duplikate) dient. Aufgrund der hohen Anzahl an notwendigen Überprüfungen auf Duplikate sowie dem dabei entstehenden ungeordneten Zugriff auf die Daten ist eine Verwendung von Festplattenspeicher ungeeignet. Um dennoch die Speicherbelastung möglichst gering zu halten, wird für die Speicherung der als Hash-Struktur organisierten Tripel-Informationen eine Adaption verschiedener Komprimierungsverfahren angewandt. Durch diese Vorgehensweise kann ein Tripel bestehend aus drei numerischen Werten mit einem Wertebereich eines 64 Bit Datentyps im Schnitt mit 6,2 Byte abgelegt werden. Aufgrund der durch die Hash-Struktur bedingten zusätzlichen Speicherbelastung erhöht sich dieser Wert in den durchgeführten Versuchen auf bis zu 10 Byte, was für den zuvor genannten Datensatz mit rund 380 Millionen Tripel in einer Gesamtspeicherbelastung von rund 3,1 GB resultiert.

Während die zuvor aufgeführten Ergebnisse zur Beantwortung der Frage beitragen, wie die massiv parallele Hardware moderner Grafikprozessoren für einen Reasoning-Prozess mit einem möglichst geringen Ressourcenverbrauch genutzt werden kann, vermittelt die durchgeführte Evaluation zusätzlich einen Eindruck von der Effektivität der eingeführten Konzepte. Besonders deutlich zeigt sich ein Unterschied in der Ausführungsgeschwindigkeit zwischen dem Reasoning unter Nutzung von GPUs, der ausschließlichen Nutzung einer CPU durch mehrere Prozesse sowie des seriellen Reasonings für die pD* Semantik. Durch die Nutzung der sechs physikalischen CPU-Prozessorkerne der verwendeten Ausführungsumgebung wird bereits gegenüber der seriellen Ausführung eine 5,3-fache Ausführungsgeschwindigkeit erreicht. Die Verwendung von zwei GPUs hingegen führt bei dem durchgeführten Versuch zu einer 86-fachen Ausführungsgeschwindigkeit gegenüber der Nutzung der CPU Ressourcen durch mehrere Prozesse. Weitere Versuche mit leichtgewichtigeren Ontologiesprachen, die insgesamt in einer gerin-

geren Anzahl an notwendigen Match-Operationen resultieren, zeigen einen entsprechend niedrigeren Performancegewinn durch die Nutzung der GPUs.

Die Kombination der zuvor genannten Aspekte der Überführung des RETE-Algorithmus für die Ausführung auf massiv paralleler Hardware, die dabei erreichte Effektivität sowie die Reduzierung des Speicherverbrauchs für die Ausführung des RETE-Algorithmus und der weiteren Schritte des Reasoning-Prozesses um rund 90%, tragen gemeinsam zur Erreichung der Zielsetzung dieser Arbeit bei. Mithilfe der eingeführten Konzepte lässt sich das Reasoning mit Large-scale-Datensätzen auf einzelnen Rechnern ausführen. Die durchgeführte Evaluation zeigt z.B. das RDFS Reasoning auf über einer Milliarde Tripel auf einem Laptop mit lediglich 16 GB Speicher. Der erreichte Durchsatz ist zum Teil höher als bei Cluster-basierten Ansätzen, die eine Vielzahl an Rechnern für die Verteilung der Arbeitslast verwenden. Ein größerer Hauptspeicher von 64 GB wird bei den Versuchen mit einer Workstation genutzt, die zusätzlich mit zwei Grafikkarten ausgestattet ist. Auf diesem Rechner wird das Reasoning auf Datensätzen mit bis zu 3,2 Milliarden Tripel durchgeführt und ein nochmals deutlich gesteigerter Durchsatz erzielt, der zum Teil die veröffentlichten Ergebnisse der verwandten Arbeiten um ein Vielfaches übersteigt.

12 FAZIT UND AUSBLICK

Die Ergebnisse der durchgeführten Evaluation zeigen eine besonders hohe Effektivität des eingeführten Konzepts der parallelen Berechnung der Match-Operationen auf massiv paralleler Hardware. Gleichzeitig verdeutlichen die durchgeführten Versuche die mögliche Rechenlast, die in Abhängigkeit des verwendeten Regelsatzes verursacht werden kann und z.B. für die pD* Regeln auf dem Lehigh University Benchmark-Datensatz mit einer Verdoppelung der Datensatzgröße zu einer Vervierfachung der Anzahl an notwendigen Operationen führt. Die Gefahr dieser als kombinatorische Explosion [AT93] bezeichneten Aufwandssteigerung, die letztendlich ausschlaggebend für die Skalierbarkeit eines Reasoning-Prozesses sein kann, ist ein Nachteil, der durch die Verwendung des RETE-Algorithmus entsteht. Aufbauend auf der Abwägung über die Komplexität eines verwendeten Regelsatzes und der damit verbundenen Rechenlast für einen Anwendungsfall, ergeben sich einige nichtfunktionale Schwerpunkte, die den in dieser Arbeit beschriebenen Forschungsstand ergänzen können.

Die Ergebnisse der durchgeführten Evaluation zeigen für das Reasoning mit den RDFS Regeln (bzw. dem Subset ρdf) insbesondere auf der verwendeten Workstation einen hohen Durchsatz. Die tatsächlich erreichte Auslastung der GPUs hingegen deutet darauf hin, dass bei den wenig komplexen Regeln (keine kombinatorische Explosion) ein noch höherer Durchsatz durch die Optimierung der Host-seitigen Verarbeitung möglich ist. Diese beinhaltet vor allem das Lesen und Schreiben der für die Ausführung notwendigen Informationen, die auf der Festplatte gespeichert werden. Entsprechend liegt diesen Informationen im Gegensatz zur Speicherung im Hauptspeicher eine wesentlich längere Zugriffszeit zugrunde. Somit kann ein Schwerpunkt für weitere Forschungsarbeiten auf einer effizienteren Bereitstellung notwendiger Daten basieren, die beispielsweise durch Verfahren der Komprimierung erreicht werden kann [WZ99]. Die Anwendung eines Komprimierungsverfahrens kann gleichzeitig zur Reduzierung der Datenmenge beitragen, die zwischen einem Host und einem Device (z.B. GPU) ausgetauscht wird. So kann eine verringerte Transferdauer und eine Dekomprimierung bzw. Komprimierung auf der massiv parallelen Hardware zu einer insgesamt gesteigerten Ausführungsgeschwindigkeit beitragen.

Eine Fokussierung auf die performante Unterstützung komplexerer Semantik ist ein weiterer Forschungsgegenstand, der basierend auf den eingeführten Konzepten vertieft werden kann. Die kombinatorische Explosion ergibt sich aus der Berechnung des Kreuzprodukts aus verschiedenen Ergebnismengen. Zur Redu-

zierung des Aufwands kann z.B. der Ansatz des *Collection oriented Match* [AT93] weiter verfolgt werden, dem eine Betrachtung der Matches in zusammengehörenden Mengen zugrunde liegt und so die Gesamtzahl der durchzuführenden Match-Operationen drastisch verringern kann. Die Herausforderung für die Umsetzung eines entsprechenden Ansatzes liegt in der Strukturierung und Organisation der Zwischenergebnisse des RETE-Algorithmus, die eine Gruppierung nach verschiedenen Eigenschaften ermöglichen müssen.

Ein weiterer Ansatz zur Erhöhung des Durchsatzes, aber gleichzeitig auch der Aufhebung der durch den Hauptspeicher beschränkten Größe von zu verarbeitenden Datensätzen stellt die Adaption der vorgestellten Konzepte z.B. auf eine MapReduce basierte Implementierung dar. Erste Arbeiten zur Überführung von regelbasierten Systemen auf eine verteilte Ausführung wurden z.B. in [CYZY10] [MOK14] und [WZLW14] veröffentlicht. Die in dieser Arbeit vorgestellten Konzepte bezüglich der Verwendung massiv paralleler Hardware für die Ausführung von Match-Operationen könnte zusätzlich zu einer effizienten Verarbeitung auf den einzelnen Rechnern eines verwendeten Clusters beitragen, so dass unter Beibehaltung der Verwendung von GPUs ein insgesamt besser skalierender Ansatz entstünde.

Während die zuvor genannten Forschungsarbeiten die Steigerung der Leistungsfähigkeit für unterschiedliche Anwendungsszenarien adressieren, ergeben sich durch einen funktionalen Ausbau des Reasoners weitere Möglichkeiten. Einerseits kann eine Erweiterung der Expressivität der Regelausdrücke eine umfangreichere Semantik zulassen und damit weitere Anwendungsfelder eröffnen. Andererseits lässt sich der entwickelte Ansatz z.B. um die Möglichkeit des Reasonings auf Daten-Streams erweitern, um auch dynamische Informationen verarbeiten zu können. Diese Ausrichtung der Erweiterung bedingt eine schnelle Verarbeitung von Daten auf einer jeweils beschränkten Menge, so dass die eingeführten Konzepte eine ideale Grundlage bieten und um Eigenschaften der Stream-Verarbeitung ergänzt werden könnten.

[Abr05] Ajith Abraham. *Rule-Based Expert Systems*. John Wiley & Sons, Ltd, 2005.

[ACZH10] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit"Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010.

[ÁGBFMP11] Sandra Álvarez-García, Nieves R. Brisaboa, Javier D. Fernández, and Miguel A. Martínez-Prieto. Compressed k2-triples for Full-In-Memory RDF Engines. In *Association for Information Systems Conference (AMCIS)*, 2011.

[AH08] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer, (Cooperative Information Systems)*. The MIT Press, 2 edition, 2008.

[AH09] Medha Atre and James A. Hendler. BitMat: A Main Memory Bit-matrix of RDF Triples. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS'09, 2009.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[Apa14] Apache Jena Project. Reasoners and rule engines: Jena inference support, 2014. Online: http://jena.apache.org/documentation/inference/index.html#rules Zuletzt besucht: 10.01.2015.

[AT93] Anurag Acharya and Milind Tambe. Collection Oriented Match. In *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, pages 516–526, New York, NY, USA, 1993. ACM.

[BACT07] Karin Breitman, Marco Antonio Casanova, and Walt Truszkowski. *Semantic Web: Concepts, Technologies and Applications*. NASA Monographs in Systems and Software Engineering. Springer, 2007.

[BC07] Stefan Büttcher and Charles L. A. Clarke. Index Compression is Good, Especially for Random Access. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management*, CIKM '07, pages 761–770. ACM, 2007.

[BCC+10] Stefano Bragaglia, Federico Chesani, Anna Ciampolini, Paola Mello, Marco Montali, and Davide Sottara. An Hybrid Architecture Integrating Forward Rules with Fuzzy Ontological Reasoning. In *Proceedings of the 5th International Conference on Hybrid Artificial Intelligence Systems*, HAIS'10, pages 438–445, Berlin, Heidelberg, 2010. Springer-Verlag.

[BCMS10] Stefano Bragaglia, Federico Chesani, Paola Mello, and Davide Sottara. A Rule-based Implementation of Fuzzy Tableau Reasoning. In *Proceedings of the 2010 International Conference on Semantic Web Rules*, RuleML'10, pages 35–49, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ber06] Berners-Lee, Tim. Linked Data, Juli 2006. Online: http://www.w3.org/DesignIssues/LinkedData.html Zuletzt besucht: 19.03.2014.

[BG00] Dan Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0: W3C Candidate Recommendation 27 March 2000. Technical report, W3C, Cambridge, MA, 2000.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, MarMar 2009.

[BHJV08] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL Reasoners. In Frank van Harmelen, Andreas Herzig, Pascal Hitzler, Zuoquan Lin, Ruzica Piskac, and Guilin Qi, editors, *Proceedings of the ARea2008 Workshop*, volume 350. CEUR Workshop Proceedings, June 2008.

[BHL+14] Johannes Busse, Bernhard Humm, Christoph Lübbert, Frank Moelter, Anatol Reibold, Matthias Rewald, Veronika Schlüter, Bernhard Seiler, Erwin Tegtmeier, and Thomas Zeh. Was bedeutet eigentlich Ontologie? *Informatik-Spektrum*, 37(4):286–297, 2014.

[BHu07] Chris Bizer, Tom Heath, and u.a. Interlinking Open Data on the Web (Poster). In *Proceedings of the 4th European Semantic Web Conference*, ESWC'07, 2007.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[BN03] Franz Baader and Werner Nutt. Basic Description Logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[Bor97] Willem Nico Borst. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, Enschede, September 1997.

[BS01] Franz Baader and Ulrike Sattler. An Overview of Tableau Algorithms for Description Logics. *Studia Logica*, 69(1):5–40, 2001.

[CHL07] Jorge Cardoso, Martin Hepp, and Miltiadis D. Lytras, editors. *The Semantic Web: Real-World Applications from Industry*, volume 6 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2007.

[CYZY10] Bin Cao, Jianwei Yin, Qi Zhang, and Yanming Ye. A MapReduce-Based Architecture for Rule Matching in Production System. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 790–795, Nov 2010.

[Dav04] Dave Beckett. W3C Recommendation: RDF/XML Syntax Specification (Revised), February 2004. Online: http://www.w3.org/TR/REC-rdf-syntax/ Zuletzt besucht: 28.04.2014.

[Den12] Andreas Dengel. *Semantische Technologien: Grundlagen. Konzepte. Anwendungen.* Spektrum Akademischer Verlag, 2012.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[DWH09] Yuxin Ding, Qing Wang, and Jiahua Huang. The Performance Optimization of CLIPS. In *Ninth International Conference on Hybrid Intelligent Systems*, volume 1, pages 417–421, Aug 2009.

[FGW84] Charles Forgy, Anoop Guptal, and Robert Wedig. Initial Assessment of Architecture for Production Systems. In *Proceedings of the National Conference for Artificial Intelligence*, pages 116–120, 1984.

[FH+08] Ernest Friedman-Hill et al. Jess, the rule engine for the java platform, 2008.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[For79] Charles Lanny Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Pittsburgh, PA, USA, 1979.

[For82] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[FU10] Florian Fischer and Gulay Unel. Reasoning in Semantic Web-based Systems. In Roberto de Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 127–146. Springer Berlin Heidelberg, 2010.

[Fur14] Frank J. Furrer. Eine kurze Geschichte der Ontologie. *Informatik-Spektrum*, 37(4):308–317, 2014.

[FvH07] Dieter Fensel and Frank van Harmelen. Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2):96, 94–95, 2007.

[GDBG05] Christine Golbreich, Olivier Dameron, Olivier Bierlaire, and Bernard Gibaud. What reasoning support for ontology and rules? The brain anatomy case study. In *Proceedings of the KR 2004 Workshop on Formal Biomedical*, pages 60–71, 2005.

[GFNW86] A. Gupta, C. Forgy, A. Newell, and R. Wedig. Parallel Algorithms and Architectures for Rule-based Systems. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 28–37, 1986.

[GHK+12] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Elsevier Science, 2012.

[GJM+11] Eric L. Goodman, Edward Jimenez, David Mizell, Sinan al Saffar, Bob Adolf, and David Haglin. High-performance Computing Applied to Semantic Databases. In *Proceedings of the 8th Extended Semantic Web Conference*, ESWC'11, pages 31–45, Berlin, Heidelberg, 2011. Springer-Verlag.

[GM10] Eric L. Goodman and David Mizell. Scalable in-memory RDFS closure on billions of triples. In *Proceedings of the sixth International Workshop on Scaleable Semantic Web Knowledge Base Systems*, SSWS'10, 2010.

[GM14] Khronos Opencl Working Group and Aaftab Munshi. The OpenCL Specification Version: 2.0 Document Revision: 22, 2014.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, October 2005.

[GQH13] Teng Gao, Xiaofeng Qiu, and Lijuan He. Improved RETE Algorithm in Context Reasoning for Web of Things Environments. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 1044–1049, Aug 2013.

[Gre11] Greenly, William and Sandeman-Craik, Charles and Otero, Yago and Streit, John. Case Study: Contextual Search for Volkswagen and the Automotive Industry, October 2011. Online: http://www.w3.org/2001/sw/sweo/public/UseCases/Volkswagen/ Zuletzt besucht: 23.03.2014.

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:199–220, 1993.

[Gup84] Anoop Gupta. Parallelism in production systems: the sources and the expected speed-up. Technical report, Computer Science Department, Carnegie Mellon University, East Lansing, Michigan, 1984.

[Gup86] Anoop Gupta. *Parallelism in Production Systems*. PhD thesis, Carnegie-Mellon University Pittsburgh, Pennsylvania, 1986.

[HJS11] Andreas Harth, Maciej Janik, and Steffen Staab. Semantic Web Architecture. In John Domingue, Dieter Fensel, and JamesA. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 43–75. Springer Berlin Heidelberg, 2011.

[HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. Springer, Berlin, 2008.

[HLTB04] Ian Horrocks, Lei Li, Daniele Turi, and Sean Bechhofer. The instance store: Description logic reasoning with large numbers of individuals. In *In International Workshop on Description Logics*, DL'04, pages 31–40, 2004.

[Hor51] Alfred Horn. On Sentences Which are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[Hor08] Ian Horrocks. Ontologies and the Semantic Web. *Communications of the ACM - Surviving the data deluge*, 51(12):58–67, December 2008.

[Hor13] Ian Horrocks. What Are Ontologies Good For? In Bernd-Olaf Küppers, Udo Hahn, and Stefan Artmann, editors, *Evolution of Semantic Systems*, pages 175–188. Springer Berlin Heidelberg, 2013.

[HP12] Norman Heino and Jeff Z. Pan. RDFS Reasoning on Massively Parallel Hardware. In *Proceedings of the 11th International Semantic Web Conference*, ISWC'12, pages 133–148, Berlin, Heidelberg, 2012. Springer-Verlag.

[IS85] Toru Ishida and Salvatore J Stolfo. Towards Parallel Execution of Rules in Production System Programs. In *Proceedings of the International Conference on Parallel Processing*, pages 568–575, 1985.

[Ish88] Toru Ishida. Optimizing Rules in Production System Programs. In *Proceedings of National Conference on Artificial Intelligence*, pages 699–704, 1988.

[Ish90] Toru Ishida. Methods and Effectiveness of Parallel Rule Firing. In *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, pages 116–122, May 1990.

[Ish94] Toru Ishida. An optimization algorithm for production systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):549–558, Aug 1994.

[Jen11] Jentzsch, Anja and Cyganiak, Richard and Bizer Chris. State of the LOD Cloud, August 2011. Online: http://lod-cloud.net/state/ Zuletzt besucht: 19.03.2014.

[JS04] Minsu Jang and Joo-Chan Sohn. Bossam: An Extended Rule Engine for OWL Inferencing. In Grigoris Antoniou and Harold Boley, editors, *Rules and Rule Markup Languages for the Semantic Web*, volume 3323 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2004.

[KB13] Tugba Kulahcioglu and Hasan Bulut. Overcoming Limitations of Term-based Partitioning for Distributed RDFS Reasoning. In *Proceedings of the Fifth Workshop on Semantic Web Information Management*, SWIM '13, pages 7:1–7:4, New York, NY, USA, 2013.

[KBM08] Vipul Kashyap, Christoph Bussler, and Matthew Moran. *The Semantic Web, Semantics for Data and Services on the Web*. Springer-Verlag, Berlin, Heidelberg, 2008.

[KKS11] Yevgeny Kazakov, Markus Krötzsch, and František Simancík. Concurrent Classification of EL Ontologies. In *Proceedings of the 10th International Conference on Semantic Web*, ISWC'11, pages 305–320, Berlin, Heidelberg, 2011. Springer-Verlag.

[KKS12] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. ELK Reasoner: Architecture and Evaluation. In *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation*, 2012.

[KKS14] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The Incredible ELK. *Journal of Automated Reasoning*, 53(1):1–61, 2014.

[KLK+14] Milhan Kim, Ki-Seong Lee, Youngmin Kim, Taejin Kim, Yunseong Lee, Sungrae Cho, and Chan-Gun Lee. RETE-ADH: an improvement to RETE for composite context-aware service. *International Journal of Distributed Sensor Networks*, 2014.

[KMK08] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *Proceedings of the 7th International Semantic Web Conference*, volume 5318, pages 499–516. Springer Berlin Heidelberg, 2008.

[KOM05] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - a Pragmatic Semantic Repository for OWL. In *Proceedings of the 2005 International Conference on Web Information Systems Engineering*, WISE'05, pages 182–192, Berlin, Heidelberg, 2005. Springer-Verlag.

[KOvH10] Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 531–540, New York, NY, USA, 2010. ACM.

[KPSH05] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging Unsatisfiable Classes in OWL Ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):268–293, December 2005.

[KR03] Joseph B. Kopena and William C. Regli. DAMLJessKB: A Tool for Reasoning with the Semantic Web. In *Proceedings of the 2nd International Semantic Web Conference*, volume 2870 of *ISWC'03*, pages 628–643. Springer Berlin Heidelberg, 2003.

[KSR+09] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media

Meets Semantic Web — How the BBC Uses DBpedia and Linked Data to Make Connections. In *Proceedings of the 6th European Semantic Web Conference*, ESWC'09, pages 723–737, 2009.

[KvHW11] Spyros Kotoulas, Frank van Harmelen, and Jesse Weaver. KR and Reasoning on the Semantic Web: Web-Scale Reasoning. In John Domingue, Dieter Fensel, and JamesA. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 441–466. Springer Berlin Heidelberg, 2011.

[KWE10] Vladimir Kolovski, Zhe Wu, and George Eadon. Optimizing Enterprise-scale OWL 2 RL Reasoning in a Relational Database System. In *Proceedings of the 9th International Semantic Web Conference*, ISWC'10, pages 436–452, Berlin, Heidelberg, 2010. Springer-Verlag.

[LB13] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[LIJ+14] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web Journal*, 2014.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[LQWY11] Chang Liu, Guilin Qi, Haofen Wang, and Yong Yu. Large Scale Fuzzy pD* Reasoning Using MapReduce. In *Proceedings of the 10th International Semantic Web Conference*, ISWC'11, pages 405–420, Berlin, Heidelberg, 2011. Springer-Verlag.

[LQWY12] Chang Liu, Guilin Qi, Haofen Wang, and Yong Yu. Reasoning with Large Scale Ontologies in Fuzzy pD* Using MapReduce. *Computational Intelligence Magazine, IEEE*, 7(2):54–66, May 2012.

[LUQ14] Chang Liu, Jacopo Urbani, and Guilin Qi. Efficient RDF Stream Reasoning with Graphics Processing Units (GPUs). In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, WWW Companion '14, pages 343–344, Republic and Canton of Geneva, Switzerland, 2014.

[LZK+09] Peiqiang Li, Yi Zeng, Spyros Kotoulas, Jacopo Urbani, and Ning Zhong. The Quest for Parallel Reasoning on the Semantic Web. In

Proceedings of the 5th International Conference on Active Media Technology, pages 430–441. Springer Berlin Heidelberg, 2009.

[Mad03] Neil Madden. Optimising RETE for low-memory, multiagent systems. In *Proceedings of the 4th International Conference on Intelligent Games and Simulation*, London, November 2003.

[May06] Wolfgang May. Reasoning im und für das Semantic Web. In *Semantic Web – Wege zur vernetzten Wissensgesellschaft*, pages 487–503, 2006.

[MB08] Georgios Meditskos and N. Bassiliades. A Rule-Based Object-Oriented OWL Reasoner. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):397–410, March 2008.

[MGMG11] Aaftab Munshi, Benedict R. Gaster, James Mattson, Timothy G. Fung, and Dan Ginsburg. *OpenCL Programming Guide*. OpenGL. Pearson Education, 2011.

[Mir87] Daniel P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 42–47. AAAI Press, 1987.

[MMH10] Raghava Mutharaju, Frederick Maier, and Pascal Hitzler. A MapReduce Algorithm for EL+. In *Description Logics'10*, 2010.

[MNP+14a] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of the Twenty-Eighth Conference on Artificial Intelligence*, AAAI'14, pages 129–137, 2014.

[MNP+14b] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel OWL 2 RL Materialisation in Centralised, Main-Memory RDF Systems. In *Informal Proceedings of the 27th International Workshop on Description Logics*, pages 311–323, 2014.

[MOK14] Ryunosuke Maeda, Naoki Ohta, and Kazuhiro Kuwabara. MapReduce-Based Implementation of a Rule System. In Amelia Badica, Bogdan Trawinski, and Ngoc Thanh Nguyen, editors, *Recent Developments in Computational Collective Intelligence*, volume 513 of *Studies in Computational Intelligence*, pages 197–206. Springer International Publishing, 2014.

[MPG07] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal Deductive Systems for RDF. In *The Semantic Web: Research and Applications*, volume 4519, pages 53–67. 2007.

[MRD+06] Carolyn J. Mattingly, Michael C. Rosenstein, Allan Peter P. Davis, Glenn T. Colby, John N. Forrest, and James L. Boyer. The comparative toxicogenomics database: a cross-species resource for building chemical-gene interaction networks. *Toxicological sciences : an official journal of the Society of Toxicology*, 92(2):587–595, August 2006.

[MRR12] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[MRS09] StuartE. Middleton, DavidDe Roure, and NigelR. Shadbolt. Ontology-based recommender systems. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 779–796. Springer Berlin Heidelberg, 2009.

[MS06] Boris Motik and Ulrike Sattler. A Comparison of Reasoning Techniques for Querying Large Description Logic ABoxes. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'06, pages 227–241, Berlin, Heidelberg, 2006. Springer-Verlag.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. Technical report, W3C, 2004.

[Nei13] Neil Trevett. OpenCL Introduction, 2013. Online: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf Zuletzt besucht: 16.04.2014.

[New73] Allen Newell. Productions systems: Models of control structures. *Visual information processing*, pages 463–526, 1973.

[NGR88] P. Pandurang Nayak, Anoop Gupta, and Paul S. Rosenbloom. Comparison of the Rete and Treat Production Matchers for Soar. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 693–698, 1988.

[NGR93] P. Nayak, A. Gupta, and P. S. Rosenbloom. The soar papers (vol. 1). chapter Comparison of the RETE and TREAT Production Matchers for Soar (a Summary), pages 621–626. MIT Press, Cambridge, MA, USA, 1993.

[Nvi09] Nvidia. OpenCL Best Practices Guide Version 1.0, 2009. Online: http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf Zuletzt besucht: 10.01.2015.

[Obe14] Daniel Oberle. Ontologies and Reasoning in Enterprise Service Ecosystems. *Informatik-Spektrum*, 37(4):318–328, 2014.

[OKA+09a] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. MARVIN: A platform for large-scale analysis of Semantic Web data. In *Proceedings of the WebScience '09*. Society On-Line, 2009.

[OKA+09b] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. MARVIN: Distributed Reasoning over Large-scale Semantic Web Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(4):305–316, December 2009.

[ÖÖÜ07] Tugba Özacar, Övünç Öztürk, and Murat Osman Ünalir. Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data. *Journal of Computers*, 2(4), 2007.

[Pan04] Jeff Z. Pan. *Description Logics: Reasoning Support for the Semantic Web*. PhD thesis, School of Computer Science, The University of Manchester, 2004.

[PB06] Tassilo Pellegrini and Andreas Blumauer. *Semantic Web: Wege zur vernetzten Wissensgesellschaft*. X. media. press Series. Springer, 2006.

[PBS12a] Martin Peters, Christopher Brink, and Sabine Sachweh. Domain Independent Architecture and Behavior Modeling for Pervasive Computing Environments. In *Proceedings of the Sixth International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS'12, pages 327–334, July 2012.

[PBS12b] Martin Peters, Christopher Brink, and Sabine Sachweh. Including Metadata into an Ontology Based Pervasive Computing Architecture. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1507–1512, June 2012.

[PBSZ13a] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Performance Considerations in Ontology Based Ambient In-

telligence Architectures. In *Ambient Intelligence-Software and Applications*, 4th International Symposium on Ambient Intelligence, pages 121–128, 2013.

[PBSZ13b] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Rule-based Reasoning on Massively Parallel Hardware. In *Proceedings of the 9th International Workshop on Scalable Semantic Web Knowledge Base Systems*, SSWS'13, pages 33–49, 2013.

[PBSZ14] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Scaling Parallel Rule-based Reasoning. In *Proceedings of the 11th Extended Semantic Web Conference*, ESWC'14, 2014.

[PGSH14] Sambhawa Priya, Yuanbo Guo, Michael Spear, and Jeff Heflin. Partitioning OWL Knowledge Bases for Parallel Reasoning. In *IEEE International Conference on Semantic Computing (ICSC)*, pages 108–115, June 2014.

[PS04] Bijan Parsia and Evren Sirin. Pellet: An OWL DL Reasoner. In *Proceedings of the International Workshop on Description Logics*, page 2003, 2004.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, W3C Recommendation, January 2008. Online: http://www.w3.org/TR/rdf-sparql-query/ Zuletzt besucht: 26.03.2014.

[PSH07] Peter F. Patel-Schneider and Ian Horrocks. A Comparison of Two Modelling Paradigms in the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(4):240–250, December 2007.

[PSZ15] Martin Peters, Sabine Sachweh, and Albert Zündorf. Large scale rule-based Reasoning using a Laptop. In *Proceedings of the 12th Extended Semantic Web Conference*, ESWC'15, 2015.

[RDF14] RDF Working Group. Resource Description Framework (RDF), February 2014. Online: http://www.w3.org/RDF/ Zuletzt besucht: 14.03.2014.

[RLB+90] Michael C. Rowe, Jay Labhart, Robert Bechtel, Steve Matney, and Steve Carrow. Forward chaining parallel inference. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 455–462, Dec 1990.

[SBF98] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25(1-2):161–197, March 1998.

[SBP14] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the Linked Data Best Practices in Different Topical Domains. In *Proceedings of the 13th International Semantic Web Conference*, volume 8796 of *ISWC'14*, pages 245–260. Springer International Publishing, 2014.

[Sca12] Matthew Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning, 2012.

[SMP08] Davide Sottara, Paola Mello, and Mark Proctor. Adding Uncertainty to a Rete-OO Inference Engine. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin Heidelberg, 2008.

[SMP10] D. Sottara, P. Mello, and M. Proctor. A Configurable Rete-OO Engine for Reasoning with Different Types of Imperfect Information. *Knowledge and Data Engineering, IEEE Transactions on*, 22(11):1535–1548, Nov 2010.

[SP08a] Ramakrishna Soma and Viktor K. Prasanna. A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases. In *21st International Conference on Parallel and Distributed Computing and Communication Systems*, pages 19–25, 2008.

[SP08b] Ramakrishna Soma and V.K. Prasanna. Parallel Inferencing for OWL Knowledge Bases. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 75–82, Sept 2008.

[SP09] Jan Schmidt and Tassilo Pellegrini. Das social semantic web aus kommunikationssoziologischer perspektive. In Andreas Blumauer and Tassilo Pellegrini, editors, *Social Semantic Web*, X.media.press, pages 453–468. Springer Berlin Heidelberg, 2009.

[SS82] Salvatore J. Stolfo and David Elliot Shaw. DADO: A Tree-Structured Machine Architecture for Production Systems. In *Proceedings of the National Conference on Artificial Intelligence. Pittsburgh*, pages 242–246, 1982.

[SS09] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. Springer, 2nd edition, 2009.

[SS11a] Anne Schlicht and Heiner Stuckenschmidt. MapResolve. In *Proceedings of the 5th International Conference on Web Reasoning and Rule Systems*, RR'11, pages 294–299, Berlin, Heidelberg, 2011. Springer-Verlag.

[SS11b] Christian Seitz and René Schönfelder. Rule-based OWL Reasoning for Specific Embedded Devices. In *Proceedings of the 10th International Conference on The Semantic Web*, ISWC'11, pages 237–252, Berlin, Heidelberg, 2011. Springer-Verlag.

[Sto84] Salvatore J. Stolfo. Five Parallel Algorithms for Production System Execution on the DADO Machine. In Ronald J. Brachman, editor, *Proceedings of the National Conference on Artificial Intelligence. Austin, Texas*, pages 300–307, 1984.

[TAFK12] Ilias Tachmazidis, Grigoris Antoniou, Giorgos Flouris, and Spyros Kotoulas. Scalable Nonmonotonic Reasoning over RDF data using MapReduce. In *Workshop on Scalable and High-Performance Semantic Web Systems*, pages 75–90, 2012.

[TBKO09] Wei Tai, R. Brennan, J. Keeney, and D. O'Sullivan. An Automatically Composable OWL Reasoner for Resource Constrained Devices. In *Semantic Computing, 2009. ICSC '09. IEEE International Conference on*, pages 495–502, Sept 2009.

[tH05] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):79–115, October 2005.

[Tim11] Tim Berners-Lee and Dan Connolly. W3C Recommendation: Notation3 (N3): A readable RDF syntax, March 2011. Online: http://www.w3.org/TeamSubmission/n3/ Zuletzt besucht: 28.04.2014.

[TKO11] Wei Tai, John Keeney, and Declan O'Sullivan. COROR: A COmposable Rule-Entailment Owl Reasoner for Resource-Constrained Devices. In *Proceedings of the 5th International Symposium Rule-Based Reasoning, Programming, and Applications*, pages 212–226, 2011.

[TKO13] Wei Tai, John Keeney, and Declan O'Sullivan. Resource-Constrained Reasoning Using a Reasoner Composition Approach. *Semantic Web Journal, (submitted, status: accepted)*, 2013.

[TKR92] M. Tambe, D. Kalp, and P.S. Rosenbloom. An efficient algorithm for production systems with linear-time match. In *Proceedings of the 4th International Conference on Tools with Artificial Intelligence*, pages 36–43, Nov 1992.

[TNR90] Milind Tambe, Allen Newell, and PaulS. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5(3):299–348, 1990.

[TS12] Jonathan Tompson and Schlachter Schlachter. An Introduction to the OpenCL Programming Model. *Pearson Education*, 2012.

[UKM+10] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri Bal. OWL Reasoning with WebPIE: Calculating the Closure of 100 Billion Triples. In *Proceedings of the 7th Extended Semantic Web Conference*, ESWC'10, pages 213–227, Berlin, Heidelberg, 2010. Springer-Verlag.

[UKM+12] Jacopo Urbani, Spyros Kotoulas, Jason Massen, Frank van Harmelen, and Henri Bal. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web*, 10, 2012.

[UKOH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Harmelen. Scalable Distributed Reasoning Using MapReduce. In *Proceedings of the 8th International Semantic Web Conference*, ISWC'09, pages 634–649, Berlin, Heidelberg, 2009. Springer-Verlag.

[UMD+13] Jacopo Urbani, Jason Maassen, Niels Drost, Frank Seinstra, and Henri Bal. Scalable RDF Data Compression with MapReduce. *Concurrency and Computation: Practice and Experience*, 25(1):24–39, January 2013.

[UMJ+13] Jacopo Urbani, Alessandro Margara, Ceriel Jacobs, Frank van Harmelen, and Henri Bal. DynamiTE: Parallel Materialization of Dynamic RDF Data. In *Proceedings of the 12th International Semantic Web Conference*, ISWC'13, pages 657–672. Springer Berlin Heidelberg, 2013.

[ÜÖÖ05] Murat Osman Ünalir, Tugba Özacar, and Övünç Öztürk. Reordering Query and Rule Patterns for Query Answering in a Rete-Based Inference Engine. In *Web Information Systems Engineering – WISE 2005 Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 255–265. 2005.

[Urb13] Jacopo Urbani. *On Web-scale Reasoning*. PhD thesis, Computer Science Deptartment, Vrije Universiteit, Amsterdam, Netherlands, 2013.

[VW10] Peter Van Weert. Efficient Lazy Evaluation of Rule-Based Programs. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1521–1534, Nov 2010.

[W3C12] W3C Recommendation. OWL 2 Web Ontology Language Profiles (Second Edition), December 2012. Online: http://www.w3.org/TR/owl2-profiles/ Zuletzt besucht: 02.04.2014.

[WED+08] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan. Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In *Proceedings of the 24th International Conference on Data Engineering*, ICDE '08, pages 1239–1248, Washington, DC, USA, 2008. IEEE Computer Society.

[WH92] Yu-Wang Wang and Eric N. Hanson. A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 88–97, Washington, DC, USA, 1992. IEEE Computer Society.

[WH09] Jesse Weaver and James A. Hendler. Parallel Materialization of the Finite RDFS Closure for Hundreds of Millions of Triples. In *Proceedings of the 8th International Semantic Web Conference*, ISWC'09, pages 682–697, Berlin, Heidelberg, 2009. Springer-Verlag.

[WWAH10] Gregory Todd Williams, Jesse Weaver, Medha Atre, and James A. Hendler. Scalable reduction of large datasets to interesting subsets. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):365 – 373, 2010.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42:193–201, 1999.

[WZLW14] Jinghan Wang, Rui Zhou, Jing Li, and Guowei Wang. A Distributed Rule Engine Based on Message-Passing Model to Deal with Big Data. *Lecture Notes on Software Engineering*, 2(3), 2014.

[XZ10] Ding Xiao and Xiaoan Zhong. Improving Rete algorithm to enhance performance of rule engine systems. In *Proceedings of the Internatio-*

nal Conference on Computer Design and Applications, volume 3, pages V3–572–V3–575, June 2010.

[YYW11] Pingle Yang, YaLei Yang, and Ning Wang. IRETE: An improved RETE multi-entity match algorithm. In *Proceedings of the International Conference on Electronics, Communications and Control*, pages 4363–4366, Sept 2011.

[ZQL+12] Zhangquan Zhou, Guilin Qi, Chang Liu, Pascal Hitzler, and Raghava Mutharaju. Reasoning with Fuzzy-EL+ Ontologies Using MapReduce. In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 933–934, 2012.

[ZWC95] Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data Partition and Parallel Evaluation of Datalog Programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):163–176, February 1995.

ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig, ohne unerlaubte Hilfe Dritter angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Dritte waren an der inhaltlich-materiellen Erstellung der Dissertation nicht beteiligt; insbesondere habe ich hierfür nicht die Hilfe eines Promotionsberaters in Anspruch genommen. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Dortmund, Juli 2015

Martin Peters