UNIVERSITÄT LEIPZIG
Fakultüt für Mathematik und Informatik
Institut für Informatik

Implementierung von Software-Frameworks am Beispiel von Apache Spark in das DBpedia Extraction Framework

Bachelorarbeit

Leipzig, 16. Mai 2018

vorgelegt von

Robert Bielinski
geb. am: 18.03.1996

Studiengang Informatik

Betreuer: Magnus Knuth
Markus Freudenberg

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, am 16. Mai 2018

Robert Bielinski

Kurzfassung

Das DBpedia-Projekt extrahiert zweimal pro Jahr RDF-Datensätze aus den semi-strukturierten Datensätzen Wikipedias. DBpedia soll nun auf ein Release-Modell umgestellt werden welches einen Release-Zyklus mit bis zu zwei vollständigen DBpedia Datensätzen pro Monat unterstützt. Dies ist mit der momentanen Geschwindigkeit des Extraktionsprozesses nicht möglich. Eine Verbesserung soll durch eine Parallelisierung mithilfe von Apache Spark erreicht werden. Der Fokus dieser Arbeit liegt auf der effizienten lokalen Nutzung Apache Sparks zur parallelen Verarbeitung von großen, semi-strukturierten Datensätzen. Dabei wird eine Implementierung der Apache Spark gestützten Extraktion vorgestellt, welche eine ausreichende Verringerung der Laufzeit erzielt. Dazu wurden grundlegende Methoden der komponentenbasierten Softwareentwicklung angewendet, Apache Sparks Nutzen für das Extraction-Framework analysiert und ein Überblick über die notwendigen Änderungen am Extraction-Framework präsentiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Zielsetzung

Das Ziel dieser Bachelorarbeit ist die Verringerung der Laufzeit des DBpedia Extraction-Frameworks durch eine Implementierung der Dump-Extraktion mithilfe des Cluster-Computing-Frameworks Apache Spark. Dabei wird der Nutzen Apache Sparks für das Extraction-Framework analysiert, die Gedanken hinter den getroffenen Implementierungsentscheidungen erläutert, die Durchführung der Implementierung an den relevanten Code-Abschnitten beschrieben und die Ergebnisse präsentiert.

1.2 Motivation

Software wiederzuverwenden ist ein wichtiger Bestandteil der Softwareentwicklung. Ein Konzept dahinter ist die Komponentenbasierte Softwareentwicklung, welche wiederverwendbare Code-Bausteine als Software-Komponenten bezeichnet. Der Vorteil dieses Konzepts ist das Potenzial, die Entwicklungszeit zu verkürzen und eine höhere Qualität der einzelnen Funktionalitäten zu gewährleisten. Die Implementierung von Software-Komponenten in bestehende Projekte kann je nach Art und Design der Komponente sehr komplex werden. Dies ist der Fall bei dem DBpedia Extraction-Framework. Die Laufzeit der Dump-Extraktion, einer schon bestehenden Funktionalität des Extraction-Frameworks, soll im Nachhinein durch eine Integration des Cluster-Computing-Frameworks Apache Spark verringert werden. Dabei soll der Nutzen und der notwendige Aufwand einer solchen Refaktorisierung ermittelt werden.

1.3 Beispielprojekt

Zunächst wird das Beispielprojekt vorgestellt. Es wird ein kurzer Überblick über DBpedia, das Extraction-Framework und Apache Spark gegeben, sowie die Notwendigkeit des Projekts und die Ziele erläutert.

1.3.1 Ausgangslage

DBpedia, ein Gemeinschaftsprojekt der Universität Leipzig und anderen Organisationen, ist eines der größten Projekte im Bereich Linked-Data. Das Ziel ist die Extraktion von Wissen in Form von RDF-Daten aus Wikipedia. Die extrahierten Daten werden online zum Download zur Verfügung gestellt und in Semantic Web Graphen geladen, die über SPARQL-Endpunkte abfragbar sind. Der Hauptteil der Daten wird durch das DBpedia Extraction-Framework aus den Wikipediadatensätzen extrahiert. Die Erstellung eines kompletten DBpedia Release dauert über einen Monat und muss in dieser Zeit dauerhaft gewartet werden. Dieser Prozess wird nur geringfügig parallelisiert, wodurch die verfügbaren Ressourcen des Extraction-Servers nicht gut genutzt werden. Skalierung durch Parallelisierung oder Cluster-Computing ist momentan nicht möglich.

Im Jahr 2017 wurde ein Projekt zur Verbesserung von DBpedia gestartet, welches DBpedia unter anderem für Fremdakteure attraktiver gestalten soll. Dazu wurden mehrere Projekte geplant und gestartet, die sich mit Aspekten wie Aktualität, Datenkonsistenz oder Integration von Fremd-Datenquellen beschäftigen. Eines dieser Projekte ist die Beschleunigung der Datenextraktion.
Da viele Nutzer von regelmäßigen Aktualisierungen der Datensätze profitieren würden, wurde der Wunsch nach kürzeren Release-Zyklen geäußert. Geplant ist der Wechsel auf einen Release-Modell welcher regelmäßige Aktualisierungen der Datensätze, bis zu zweimal pro Monat vorsieht. Dadurch wird der von DBpedia zur Verfügung gestellte Wissensgraph aktueller, flexibler und somit attraktiver für die Nutzer. Um diese kürzeren Release-Zyklen zu ermöglichen, muss jedoch die Extraktion der Daten erheblich beschleunigt und automatisiert werden. Diese Bachelorarbeit beschäftigt sich mit dieser Beschleunigung, welche durch eine Apache Spark unterstützte Implementierung der Extraktion ermöglicht werden soll.

1.3.2 Zielsetzung

Das Ziel ist die Beschleunigung des Datenflusses im Extraction-Framework mit Hilfe von Apache Spark. Im Laufe dieser Bachelorarbeit soll dies für die zentralen Software-Komponenten des Extraction-Frameworks umgesetzt werden. Als konkrete Anforderung wurden zu Beginn drei der Extractor-Klassen, die bestimmte Funktionalitäten der Anwendung realisieren, ausgewählt, um als Leitlinie der Implementierung zu dienen. Zuerst wird gezeigt, dass der allgemeine Extraktions-Fluss mit Unterstützung Apache Sparks funktioniert und anschließend, dass auch komplexere Funktionen umsetzbar sind. Dabei sollen alle notwendigen Änderungen an den Extractor-Klassen dokumentiert werden.

1.4 Überblick über diese Arbeit

Als Erstes werden in dem Kapitel *Grundlagen* alle verwendeten Begriffe und Methoden definiert und alle verwendeten Komponenten ausführlich beschrieben.
Anschließend wird in *Konzeption* die Durchführung des Projekts geplant, die genauen Anforderungen beschrieben, ein erster Prototyp implementiert, ausgewertet und ein genauer Plan für die Phase der Implementierung aufgestellt. Es werden alle relevanten Aspekte Apache Sparks im Hinblick auf diese Implementierung analysiert und nach dem

Aspekt der Nützlichkeit bewertet.
Das folgende Kapitel *Implementierung* beschäftigt sich mit der konkreten Umsetzung der einzelnen Arbeitspakete. Alle größeren Probleme werden analysiert und die gefundenen Lösungsansätze an relevanten Code-Abschnitten präsentiert. Außerdem werden hier alle erforderlichen Änderungen an die Extractor-Klassen des Extraction-Frameworks definiert, was insbesondere für die Entwickler der DBpedia-Community interessant ist. Im Kapitel *Ergebnisse* wird die Implementierung in Hinsicht auf die Erfüllung der Anforderungen getestet. Es werden Laufzeit- und Datensatzvergleiche durchgeführt und ihre Ergebnisse präsentiert. Zusätzlich wird ein Überblick über den Gesamtfortschritt der Anpassung gegeben und der notwendige Restaufwand eingeschätzt.
Schließlich werden in *Fazit* die Ergebnisse zusammengefasst und ein Ausblick über mögliche zukünftige Weiterarbeit und Fertigstellung des Projekts gegeben.

Kapitel 2

Grundlagen

2.1 Komponentenbasierte Softwareentwicklung

Im Lehrbuch der Softwaretechnik von H. Balzert [1] werden verschiedene Definitionen für den Begriff *Software* gegeben. Diese Definitionen fassen Software als Sammelbegriff für alle Programme, einschließlich ihrer Dokumentation, auf, die auf Rechensystemen ausgeführt werden können. Die Softwareentwicklung hat im Laufe der Jahre verschiedene Herangehensweisen zutage gebracht. Wurden zu Beginn noch Einsen und Nullen geschrieben, wurde der Prozess immer weiter abstrahiert, bis man heutzutage fertiggestellte Komponenten zusammenfügt und nur noch die Schnittstellen verbindet. Das Themengebiet der *komponentenbasierten Softwareentwicklung* beschriebt genau diese Herangehensweise. Demnach wird Software als Produkt von Software-Komponenten (auch Halbfabrikate genannt) angesehen, die nach H. Balzert [1] als abgeschlossene, binäre Software-Bausteine definiert werden. Diese stellen eine anwendungsorientierte, semantisch zusammengehörige Funktionalität nach außen hin über Schnittstellen zur Verfügung. Eine Definition die sehr viel Raum für unterschiedliche Auslegungen lässt.
In Componentware [2] wird eine Klassifizierung nach Abstraktheitsgrad gegeben. Die *white-box*-Komponenten sind Konzepte, die nur wenig Code besitzen. Verwendet werden sie durch Vererbung der bereitgestellten Klassen und Implementierung ihrer Methoden. Das Gegenstück dazu sind die *black-box*-Komponenten, die konkrete Nutzung ihrer Methoden über Schnittstellen zur Verfügung stellen.
Verwandt mit den Software-Komponenten sind die Software-Frameworks. Sie geben einen Architekturrahmen für die Software-Komponenten vor. Frameworks lassen sich ebenso je nach Abstraktheitsgrad als *White-box-* oder als *Black-box*-Framework klassifizieren. White-box-Framworks sind Abstrakte Klassen, welche durch Vererbung implementiert werden. Die abstrakten Klassen besitzen bestimmte Atrribute oder Relationen zu anderen Klassen, wodurch ein Architekturrahmen festgelegt wird. Black-box-Frameworks bestehen hingegen aus mehreren Software-Komponenten die zueinander in Beziehung stehen. Sie werden von außen hin über Schnittstellen angesprochen und definieren „Lücken“ in welche andere Komponenten eingesetzt werden sollen. Stellt ein Framework statt eines einzelnen Lösungsentwurfs ein vollständiges Anwendungssystem zur Verfügung, wird es auch als *Application Framework* bezeichnet[3][4].
Außerdem lassen sich Software-Frameworks auch nach Anwendungsdomäne klassifizie-

ren. Unter anderem gibt es Web-Frameworks wie zum Beispiel Ruby on Rails oder Test-Frameworks wie JUnit.
Das allgemeine Vorgehen zur komponentenbasierten Entwicklung von Software ist das Zusammenfügen von Softwarekomponenten. Durch das Fehlen von Standards ist es schwer eine allgemeingültige Vorgehensweise zu definieren. In Software-Engineering: Software Components [5] wird das Problem in drei Bereiche einteilt: Domain, Connectors und Behaviour. Domain beschreibt dabei die Domäne der Komponenten. Durch Bestimmung dieser im Domain-Engineering wird es möglich, Gemeinsamkeiten der Komponenten darzustellen und gut kooperierende Komponenten auszuwählen. Connector beschäftigt sich mit den Schnittstellen der Komponenten. Hier werden die Komponenten zusammengefügt und der Zwischenraum zwischen den Schnittstellen gefüllt. Der letzte Teil ist das Behaviour, also das Verhalten der Komponente. Hier wird die Komponente an das vorliegende Problem, durch Konfiguration oder Anpassung des Codes angepasst.
Schwieriger wird die Integration eines Software Frameworks in ein schon bestehendes Projekt. Von außen hin sind Black-box-Frameworks vergleichbar mit Komponenten. Man muss also auch hier wieder die Damäne, die Schnittstellen und das Verhalten betrachten. Nur besitzen sie zusätzlich vordefinierte innere Schnittstellen, an denen man andere Komponenten einsetzen muss.
Will man eine schon vorhandene Funktionalität durch ein Framework ersetzen, müssen sowohl die verwendeten inneren Komponenten als auch die Komponenten, die von außen mit dem Framework interagieren mit dem Framework kompatibel sein und in Zusammenarbeit die gleichen Anforderungen erfüllen.

2.2 Linked Data

In diesem Abschnitt wird ein Überblick über das Themengebiet Linked-Data erarbeitet. Es wird der Begriff Semantic Web definiert und wichtige Komponenten und Formate kurz beschrieben.

2.2.1 Semantic Web

Das Semantic Web ist eine Erweiterung des World Wide Web um eine semantische Komponente, die Computern dabei hilft, das Web besser nutzen zu können [6]. Computern, beziehungsweise Softwareanwendungen, wird ermöglicht relevante Informationen in einem Text zu erkennen und automatisch zueinander in Beziehung zu setzen[6]. Dies wird zum Beispiel von Suchmaschinen genutzt, um genauere Ergebnisse zu finden.

2.2.2 XML

Ein Standard zur Beschreibung von Inhalten und zum Datenaustausch ist die Extensible Markup Language **XML** [7]. XML Dateien besitzen einen Prolog- und einen Datenbereich. Im Prolog werden die XML Version, das Stylesheet und gegebenenfalls das benutzte Datenschema festgelegt. Der Datenbereich eines XML Dokuments lässt als Baum darstellen, dessen Knoten als **Elemente** bezeichnet werden. Jedes Element wird durch einen Start-Tag `<TAGNAME>` und einen End-Tag `</TAGNAME>` definiert. XML Elemente können *Attribute* der Form `<TAGNAME ATTRIBUTNAME="WERT">` besitzen.
Anders als in HTML werden die Tags der Inhalte in XML nicht zur Definition der Darstellung der Information verwendet, sondern zur Gliederung und Beschreibung der Informationen, sodass sowohl Mensch als auch Computer die Information genauer interpretieren können [8]. XML Dokumente müssen *wohlgeformt* sein und, sofern sie ein Schema definiert haben, diesem entsprechen.
XML ist weit verbreitet und besitzt viel Unterstützung in verschiedensten Anwendungen. Für das Semantic Web reicht XML allein jedoch nicht aus, da den Element- oder Atrributnamen keine Semantische Bedeutung zugeschrieben werden kann.

2.2.3 RDF

Das Resource Description Framework (kurz: **RDF**) ist ein essenzieller Bestandteil des Semantic Web und ist ein Standard zum Datenaustausch im Web. Es erlaubt die Erweiterung von Datensätzen durch einfache Verlinkung zu Fremddatensätzen, selbst wenn die grundlegenden Schemata unterschiedlich sind [9]. Somit können Informationen zwischen Applikationen ohne Informationsverlust ausgetauscht werden [9]. RDF implementiert ein einfaches Datenmodell in dem die Ressource als Subjekt über ein Prädikat mit einem Objekt verknüpft wird. Subjekte und Prädikate werden dabei über Uniform Ressource Identifier (kurz: **URI**) identifiziert, Objekte können neben einer URI auch ein Literal sein. Diese Subjekt-Prädikat-Objekt Tripel bilden zusammen einen gerichteten Graphen. Zur Repräsentation von RDF gibt es verschiedener gleich mächtiger Formate:

N-Triples

N-Tripels ist wohl der einfachste Standard der RDF Repräsentation. URIs werden in spitze Klammern {'<', '>'} gehüllt und pro Zeile des Dokuments steht ein Subjekt, ein Prädikat und ein Objekt gefolgt von einem Punkt [10].

Turtle

Turtle erweitert N-Triples durch Präfixe und anderen Konstrukten zur Vermeidung von Redundanz. Zum Beispiel kann die URI `https://www.dbpedia.org/ressource/` nach vorheriger Definition, durch einen Präfix wie zum Beispiel `dbr:` abgekürzt werden. Außerdem kann Redundanz vermieden werden in dem man durch Nutzung eines Semikolons Prädikatlisten erstellt (Analog: Komma und Objektlisten) [11].

Weitere Formate

Weitere Repräsentationen sind zum Beispiel RDF/XML, welches festlegt, wie man RDF in validem XML darstellt, oder RDFa welches RDF in HTML Dokumente integriert.

2.2.4 Web-Ontology

Um die Informationen des Semantic Web auch Interpretieren zu können braucht man zusätzlich zur Syntax, die RDF vorgibt auch ein Vokabular, eine Grammatik und Regeln für die Verwendung des Vokabulars [6]. Diese werden in einer sogenannten *Ontologie* definiert. Für diese gibt es zwei Standard Formate RDFS und OWL.
Das Ressource Description Framework Schema (kurz: **RDFS**) ist ein Schema für RDF, welches semantische Bedeutungen und Verwendungsregeln für das RDF Vokabular formuliert. Mit RDFS lassen sich einfache Ontologien modellieren.
Die Web Ontology Language (kurz: **OWL**) ist ausdrucksstärker als RDFS und basiert auf der Prädikatenlogik.

SPARQL

RDF Graphen können über sogenannte SPARQL-Endpunkte durch die Nutzung der graphenbasierten Abfragesprache SPARQL angefragt werden. SPARQL versucht SQL mit RDF zu verbinden und ist seit 2008 offiziell von W3C zum Standard erklärt worden.

2.3 bzip2

Bzip2 ist ein freies Kompimierungsprogramm welches effiziente und verlustfreie Komprimierung von Daten ermöglicht [1]. Seit 2003 unterstützt es auch paralleles komprimieren und dekomprimieren [12]. Dies wird durch die Nutzung mehrerer Ausgabeströme ermöglicht, welche im Anschluss konkateniert werden. Diese Ströme können anschließend auch wieder zur parallelen Dekomprimierung genutzt werden.

[1] bzip2: https://de.wikipedia.org/wiki/Bzip2

2.4 Wikipedia

Wikipedia ist ein gemeinnütziges Projekt der Wikimedia Foundation zur Erstellung einer freien Online Enzyklopädie[2]. Die Artikel werden von freiwilligen Autoren geschrieben und kontrolliert. Über 41 Millionen Artikel erstrecken sich inzwischen über bis zu 295 Sprachen [3]. Die Artikel nutzen häufig vorgefertigte Templates, wiederverwendbare Artikel-Komponenten, die es leicht machen einheitliche Artikel zu schreiben und somit auch leichter bestimmte Informationen zu extrahieren [4].
Neben der Wikipedia Webseite stehen zur Beschaffung der Daten auch die Wikipedia API [5], die Wikidata Webseite, als auch die Wikimedia Dump Downloads [6] zur Verfügung. Die Wikipedia API ist jedoch nur für gezielte Anfragen gedacht. Zur großflächigen Verarbeitung der Daten werden die Wikipedia Dumps empfohlen.
Die Wikipedia Artikel Dumps werden in verschiedenen Formaten angeboten. Die XML-Darstellung lässt sich mit dieser vereinfachten Dokumentstruktur veranschaulichen:

```
1  <mediawiki>
2      <page>
3          <title>
4          <ns>
5          <id>
6          <revision>
7              <id>
8              <timestamp>
9              <contributor>
10                 <username>
11                 <id>
12         <text>
13         <format>
```

Listing 2.1: Wikipedia-Dump-XML vereinfachte Baumstruktur

Die Wurzel des XML-Baums ist das `<mediawiki>`-Element und die einzelnen Seiten werden durch `<page>`-Elemente dargestellt. Eine Seite besitzt einen Titel `<title>`, einen Namespace `<ns>`, der angibt um was für eine Art von Artikel es sich handelt, eine ID `<id>`, eine Revision `<revision>`, den Artikeltext `<text>` und ein Format `<format>`. Das Revisionselement enthält Daten über den aktuellen Stand des Artikels, wann und von wem er zum letzten Mal geändert wurde.

2.5 DBpedia

DBpedia ist ein Open-Source-Projekt der Universität Leipzig, der Universität Mannheim, Open Link Software und dem Hasso-Plattner Institut [13]. Das Ziel ist die Extraktion von Informationen aus Wikipedia und die Darstellung dieser Information in RDF-Form in einem Semantic Web Graphen, welcher über SPARQL-Endpunkte an-

[2] Wikipedia: https://de.wikipedia.org/wiki/Wikipedia
[3] Wikipedia Größenvergleich: https://de.wikipedia.org/wiki/Wikipedia:Gr%C3%B6%C3%9Fenvergleich
[4] Wikipedia Templates: https://en.wikipedia.org/wiki/Help:A_quick_guide_to_templates
[5] Wikipedia API: https://www.mediawiki.org/wiki/API:Main_page/de
[6] Wikipedia Datadumps: https://dumps.wikimedia.org/

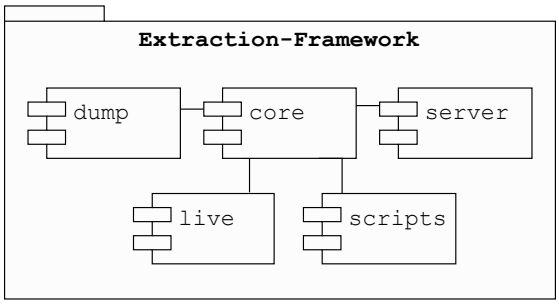

Abbildung 2.1:
Extraction-Framework Modulübersicht

fragbar ist.
Der Hauptteil der Datenextraktion wird durch das DBpedia Extraction-Framework [7] auf den Wikipedia Dump-Datensätzen durchgeführt. Zusätzlich werden bei manchen Extraktionen auch die Wikipedia API oder die Wikidata JSON-Dumps zur Extraktion von RDF-Daten genutzt.
DBpedia stellt die Plattform für viele verschiedene Projekte, wie zum Beispiel DBpedia Spotlight[8] [14], welches automatische Textannotation von DBpedia Ressourcen ermöglicht.

2.6 DBpedia Extraction-Framework

Das **DBpedia Extraction-Framework** ist ein essenzieller Bestandteil von DBpedia. Die Anwendung ist in den Programmiersprachen Java und Scala geschrieben und verfolgt das Ziel RDF Daten aus den verschiedenen Datenquellen Wikipedias zu extrahieren. Diese extrahierten Daten sind die Basis für die RDF-Graphen und Downloads auf DBpedia.
Java ist eine weitverbreitete, objektorientierte Programmiersprache, deren Programme auf einer Virtuellen Maschine, der *JVM*, ausgeführt, mithilfe der Entwickler-Werkzeuge, der *JDK*, entwickelt und unter Verwendung der Java Laufzeit Umgebung, der *JRE* ausgeführt werden. Nach dem TIOBE-Index [9] befindet sich Java aktuell (Stand April 2018) auf Platz 1 der populärsten Programmiersprachen; weiter abgeschlagen befindet sich die Programmiersprache **Scala** nur auf dem 34. Platz. Scala steht für "Scalable Language" und ist eine objektorientierte Programmiersprache, die auf Java aufbaut und komplett mit Java kompatibel ist [15]. Ziel von Scala ist die Einbringung vieler funktionaler Herangehensweisen wie Lambda-Ausdrücken oder der Typinferenz. Die native Unterstützung dieser funktionalen Elemente und das Weglassen nicht unbedingt notwendiger Klammern ermöglichen kürzeren und übersichtlicheren Programmcode als Java.
Das Extraction-Framework ist ein Application Framework. Es ist eine ausführbare Anwendung die über verschiedene Schnittstellen erweiterbar ist. Die wichtigste Schnitt-

[7] DBpedia Extraction-Framework: https://github.com/dbpedia/extraction-framework
[8] DBpedia Spotlight: https://github.com/dbpedia-spotlight/dbpedia-spotlight
[9] TIOBE-Index: https://www.tiobe.com/tiobe-index//

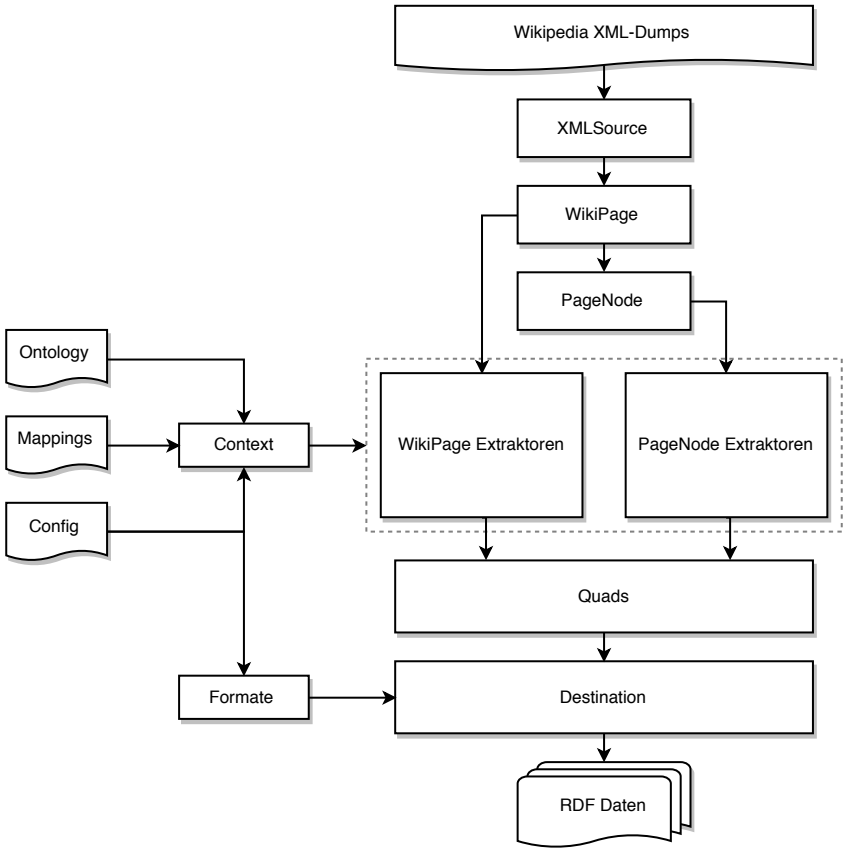

Abbildung 2.2: Datenfluss: Dump-Extraction

stelle ist die abstrakte Klasse `Extractor`. Klassen die von Extractor erben führen die Extraktion von RDF Daten aus einem Wikipedia-Artikel durch und werden als **Extraktoren** bezeichnet. Das Extraction-Framework enthält aktuell (Stand März 2018) über 60 verschiedene Extraktoren. Welche Extraktoren auf einem Datensatz ausgeführt werden sollen, ist dabei für jede Sprache frei konfigurierbar.
Das Extraction Framework besteht aus den in Abbildung 2.1 dargestellten fünf großen Modulen. Die relevanten Module für diese Bachelorarbeit sind die Module `core` und `dump`. Das core-Modul enthält alle Extraktoren und Hilfsklassen die von den anderen Modulen verwendet werden und bildet die Basis oder den Kern des Extraction-Frameworks. Das dump-Modul führt die **Dump-Extraktion** durch und enthält Klassen, welche die Konfiguration laden, die benötigten Datenesätze gegebenenfalls herunterladen und anschließend die Extraktion auf diesen Datensätzen starten.
Weitere Module sind `server`, `scripts` und `live`. Das server-Modul startet eine Serverversion des Extraction-Frameworks. Diese besitzt eine Weboberfläche, über welche man dem Extraction-Framework einzelne Wikipedia Artikel zur Extraktion geben kann. Das scripts-Modul stellt eine Vielzahl von Skripten zur Nachbearbeitung der Datensätze bereit und das live-Modul führt kleine Extraktionen über die Wikipedia API durch.

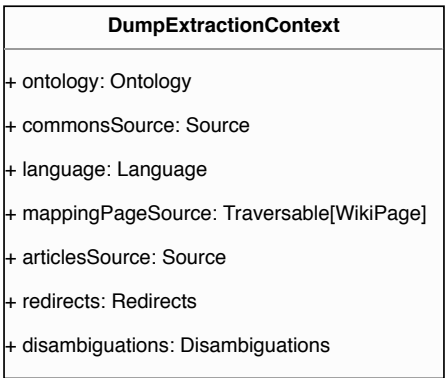

Abbildung 2.3: Dump-Extraction-Context

Das Extraction-Framework bietet viele Möglichkeiten zur Konfiguration. Einerseits gibt es die `universal.properties` im core Modul, in der einige allgemeine Einstellungen zu Pfaden, Versionen oder Präfixen angepasst werden können. Andererseits gibt es `.properties` Dateien für die ausführbaren Module des Extraction-Frameworks. Für die Dump-Extraktion gibt es eine Konfigurationsdatei, in der man festlegen kann welche Extraktoren für welche Sprache ausgeführt werden sollen. Diese Konfigurationen werden im Fall der Dump-Extraktion durch die Klasse ConfigLoader geladen und an das Extraction-Framework gegeben.
In Abbildung 2.2 wird der Datenfluss der Dump-Extraction dargestellt. Zu Beginn werden die Konfigurationen von der ConfigLoader Klasse geladen und die Extraktion initialisiert. Die XML Dumps werden von der Klasse XMLSource gelesen und zu **WikiPage**-Objekten geparst. WikiPage-Objekte sind die interne Repräsentation eines Wikipedia Artikels im Extraction-Framework und sind analog zum Schema des XMLs aufgebaut. Neben den relevanten Metadaten, enthält eine WikiPage-Instanz auch den Artikeltext geparst als Baumstruktur, der sogenannten **PageNode**. Diese wird von einigen Extraktoren verwendet, um effizient bestimmte Abschnitte des Artikels zu verarbeiten. Diese Extraktoren werden als PageNode-Extraktoren bezeichnet und führen aufwendigere Analysen und Extraktionen auf dem Artikelinhalt aus. Dagegen arbeiten WikiPage-Extraktoren auf den WikiPage Instanzen selbst und extrahieren Metadaten. Alle Extraktoren benötigen zusätzlich zu den Eingabedaten ein **Context**-Objekt, welches Daten enthält, die zur Generierung der RDF-Daten genutzt werden.

Eine Context Instanz (Abbildung 2.3) wird bei der Initialisierung der Extraktion mit folgenden Informationen gefüllt:

Ontology:
Die Ontologie ermöglicht den Extraktor-Klassen, die Vokabeln der Ontologie zu nutzen, um valides RDF zu erstellen.

Language:
Die Sprache, auf der die Extraktion ausgeführt wird, ermöglicht den Extraktor-Klassen die richtigen Mappings, Templates und Parser auszuwählen und die extrahierten Daten

der richtigen Sprache zuzuordnen.

ArticleSource:
Der Eingabestrom aus WikiPage-Instanzen, welcher von dem XMLSource-Objekt zur Verfügung gestellt und von den Extraktoren abgearbeitet wird.

Redirects:
Verweise auf die Weiterleitungen zwischen den Artikeln.

Disambiguations:
Liste an Disambiguierungen, welche vom DisambiguationsExtractor zur Auflösung von Mehrdeutigkeiten benutzt wird.

Mappings:
Mappings die von dem MappingsExtractor benutzt werden.

CommonsSource:
Wikipedia-Commons Datensatz, der bei Ausführung des ImageExtractors benutzt wird, um das Copyright der Bilder zu überprüfen.

Extraktoren werden durch die `extract` Methode ausgeführt, welche eine Liste aus **Quad**-Objekten zurückgibt. Quads sind die interne Repräsentation von RDF Daten des Extraction-Frameworks. Die über 60 Extraktoren im Extraction-Framework extrahieren oder analysieren alle unterschiedliche Aspekte der Wikipedia Artikel. Der größte Anteil an Extraktoren sind PageNode-Extraktoren und der am häufigsten verwendete Extraktor ist der MappingExtractor. Dieser wird auf den Datensätzen jeder Sprache ausgeführt und bekommt durch den Context Mappings als zusätzliche Eingabe. Diese werden für jede Sprache aus einer separaten XML-Datei gelesen. Abschließend werden die Quads an die Destination Klasse gegeben. Diese ist für die Datenausgabe des Extraction-Frameworks zuständig. Hier werden die Quads in die gewünschten RDF-Repräsentationsformen serialisiert, die Quads nach Ziel-Datensatz getrennt und in BZip2-komprimierte Dateien geschrieben.

2.7 Apache Spark

Apache Spark (kurz: **Spark**), ein Open-Source Projekt, welches 2009 als Forschungsprojekt im UC Barkeley RAD Lab gestartet wurde, ist eine Cluster-Computing Plattform die entworfen wurde um *Geschwindigkeit* mit *Allgemeingültigkeit* zu verbinden[16]. *Cluster-Computing* bezeichnet dabei die Nutzung von üblicherweise über ein LAN-Netzwerk verbundenen Rechnern die gemeinsam aufwendige Aufgaben und Berechnungen durchführen. Apache Spark setzt auf ein Master/Worker Prinzip[17]. Dies bedeutet, dass die Befehle von einem Knoten im Cluster-Netz an alle anderen Knoten gesendet werden. Der bestimmende Knoten ist der **Master**, die ausführenden Knoten die **Worker**. Das Logging ist in Apache Spark über den Apache Logging Service Log4j realisiert. Apache Spark verallgemeinert das durch Hadoop oder Google bekannte MapReduce Modell indem es beliebe Arten von Transformationen erlaubt. Bei MapReduce werden aufwendige verteilte Berechnungen in drei Schritte geteilt: `map`, `shuffle` und `reduce`. Spark fügt eine Reihe weiterer Operationen hinzu und ermöglicht voneinander unabhängige Nutzung der verschiedenen Methoden. Dies bricht das starre MapReduce Modell und ermöglicht somit eine Vielzahl an weiteren Einsatzmöglichkeiten. Berechnungen werden im Hauptspeicher durchgeführt und Datensätze können sowohl im Hauptspeicher als auch auf der Festplatte zwischengespeichert werden. Standard Anwendungen für Spark sind klassische MapReduce Algorithmen, Echtzeitdatenverarbeitung aber auch große Datenanalysen oder Machine-Learning-Algorithmen.
Implementierungen in verbreiteten Sprachen wie Java, Scala, Python und R, einfache Integration mit Hadoop und sehr gute Ergebnisse sowohl in lokaler Nutzung als auch auf verteilten Cluster- oder Cloudarchitekturen machen Apache Spark zu einer beliebten Lösung für aufwendige verteilte Berechnungen. Deswegen wird Apache Spark von vielen großen Organisationen verwendet, unter anderem von Amazon, eBay Inc. und Yahoo! [10].

2.7.1 Spark Core

Spark-Core ist wie der Name schon sagt das Kernpaket von Apache Spark. Hauptbestandteil des Core-Pakets sind die Resilient Distributed Datasets, kurz **RDD**. RDDs sind partitionierte und parallele Kollektionen, die beliebige Arten von Objekten beinhalten können. Transformationen über RDDs werden durch die Worker ausgeführt. RDDs wurden unter dem *lazy*-Ansatz entworfen, wodurch die Daten erst geladen und verarbeitet werden, wenn sie auch wirklich gebraucht werden. Für die RDDs bedeutet dies, dass man so viele Transformationen auf einer RDD definieren kann wie man möchte, sie werden jedoch erst durchgeführt wenn eine anschließende Aktion ausgeführt werden soll. Transformationen sind dabei unter anderem Eingabe-, Mapping- oder Filter-Operationen, Aktionen hingegen sind Ausgabe-, Zähl-, oder Reduce-Operationen. Apache Spark bietet die Möglichkeit Daten von der Festplatte oder aus dem Hauptspeicher in RDDs zu laden und analog die Daten der RDDs auch in den Hauptspeicher oder auf die Festplatte auszugeben.
Ein wichtiges Thema bei RDDs ist die Persistenz. Mit der Persistenz kann der Stand einer RDD zu einem beliebigen Zeitpunkt zwischengespeichert werden und kann somit

[10] Powered-By Apache Spark: https://spark.apache.org/powered-by.html

für zukünftige Nutzung wiederverwendet werden. Ohne die Nutzung von Persistenz werden alle zusammenhängenden Transformationen bei erneuter Ausführung einer Aktion neu berechnet.
Ein weiteres interessantes Thema von Apache Spark, welches im Core-Paket wichtig wird, ist die Serialisierung. Als Serialisierung wird die Umwandlung von Objekten einer Anwendung zu Bytes bezeichnet. Das Ziel ist die Konvertierung zu Bytes und anschließende Deserialisierung zu einer internen Objekt-Instanz ohne dabei Informationen zu verlieren. Spark benötigt diesen Vorgang um Informationen zwischen dem Master und den Workern auszutauschen. Dies Umfasst jegliche Daten die zwischen den Knoten versendet werden, sowie den Programmcode den die Worker ausführen sollen. Serialisierung wird von Scala durch Java nativ unterstützt und durch einfache Klassenvererbung der `Serializable`-Schnittstelle umgesetzt. Alternativ bietet Apache Spark auch die Möglichkeit das Serialisierungs-Framework **Kryo** zu verwenden. Es ist im Core-Paket enthalten und wirbt damit sehr viel schneller und ressourcensparender als die Java-Serialisierung zu sein.
Das Core-Paket wird über das SparkContext Objekt initialisiert welches vielfältig konfiguriert werden kann und die Schnittstelle für das Verteilen von Daten und die Erstellung von RDDs bietet.

2.7.2 Spark Erweiterungen

Neben dem Core-Paket stellt die Apache Software Foundation auch noch andere zusätzliche Erweiterungen für Apache Spark zur Verfügung.

Spark Streaming ist eine Erweiterung für Apache Spark, welche die Verarbeitung von Eingabeströmen aus verschiedenen Quellen ermöglicht. Dabei werden die Daten *nicht* direkt verarbeitet wie zum Beispiel bei Apache Flink, sondern erst im Hauptspeicher gesammelt und in kurzen, konfigurierbaren Abständen verarbeitet. Das Einsatzfeld für Spark Streaming sind dauerhaft laufende Anwendungen, wie zum Beispiel Live-Feed Analysen, die dauerhaft neue Eingaben aus externen Quellen verarbeiten. Spark Streaming unterstützt dabei auch Apache Kafkas Streams.

Die Daten werden zwischengespeichert und nach einem selbst definierten Zeitraum zur Verarbeitung an die Worker-Knoten gegeben. SparkSQL ist eine Erweiterung für Apache Spark, welche die Verarbeitung von strukturierten oder semi-strukturierten Daten effizienter gestaltet. Es arbeitet anders als spark-core nicht mit RDDs, sondern mit den eigenen Kollektionstypen **Dataset** und **Dataframe**. Die Serialisierung der Elemente wird durch spezielle Encoder-Klassen vorgenommen und benötigt eigene Encoder für jeden Datentyp. Nach einlesen des Datensatzes, wird dieser für die Ausführung von häufigen Anfragen optimiert. Auf Datasets und Dataframes kann man wie auf RDDs arbeiten, zusätzlich aber auch SQL-Anfragen ausführen. Seit Version 2.1 bietet SparkSQL auch Structured Streaming als Eingabemethode welches den Streaming Ansatz von Spark Streaming mit der effizienten Verarbeitung strukturierter Daten von SparkSQL verbindet.

Kapitel 3

Konzeption

In diesem Kapitel wird die Implementierung konzeptioniert. Dazu werden zunächst die Anforderungen an das Projekt spezifiziert, Apache Sparks Nutzen für das Extraction-Framework analysiert und anschließend ein erster Prototyp entwickelt.

Bevor ein Plan für die Implementierung erstellt werden kann, müssen zunächst die Anforderungen an das Projekt definiert werden. Nach einem Gespräch mit Sebastian Hellmann und seinem KILT-Team haben sich folgende Anforderungen für das Projekt ergeben:

- **1:** Apache Spark soll in den Prozess der Dump-Extraktion des Extraction-Frameworks integriert werden.
- **2:** Der LabelExtractor und der InfoboxExtractor sollen in der Apache Spark-Integration funktionstüchtig sein.
- **3:** Die Dump-Extraktion soll vor und nach der Apache Spark-Integration die gleichen Daten extrahieren (→ Datenvollständigkeit).
- **4:** Die Flexibilität der Konfigurationen des Extraction-Framework soll erhalten bleiben.
- **5:** Die verfügbaren Ressourcen des Extraction-Servers sollen besser genutzt werden.
- **6:** Durch die bessere Ressourcennutzung soll bei der Ausführung auf dem Extraction-Server die Laufzeit der Dump-Extraktion, um mindestens 60% gesenkt werden.
- **7:** Die ursprüngliche Extraktion soll noch ausführbar sein.
- **8:** Nach erfolgreicher Implementierung der beiden Extraktoren, soll eine Einschätzung über den Aufwand und die notwendigen Kenntnisse der restlichen Implementierung erstellt werden.

Die Extraktoren wurden nach genauerer Betrachtung des Extraction-Frameworks aus folgenden Gründen ausgewählt:

LabelExtractor

Der LabelExtractor wurde ausgewählt, weil er einen der einfachsten Aspekte der Extraktion umsetzt: die Extraktion des Titels der Wikiseite. Durch die Einfachheit dieser Klasse wird schnelles Testen und Ausführen des Extraction-Frameworks ermöglicht, ohne dabei auf die Einzelheiten der Extraktoren eingehen zu müssen, dies ist besonders wichtig, wenn man zuerst die allgemeinen Teile des Datenflusses umsetzen will.

InfoboxExtractor

Der InfoboxExtractor etrahiert alle relevanten Informationen aus den sogenannten Infoboxen der Wikipedia Artikel. Diese Infoboxen enthalten wichtige allgemeine Attribute und meist ein Bild der durch den Artikel beschriebenen Entität. Er gehört zur komplexeren Sorte der Extractor-Klassen. Er benötigt einige zusätzliche Daten und verwendet Parser um vielfältige Ergebnisse aus den Infoboxen der Artikel zu extrahieren. Somit ist er eine gute Wahl um zu demonstrieren, wie aufwendig die notwendigen Änderungen an den Extraktoren sind. Dies ermöglicht eine Einschätzung des Gesamtaufwandes.

MappingExtractor

Der MappingExtractor ist einer der wichtigsten Extraktoren des Extraction-Frameworks. Er ermöglicht die Anwendung sprachspezifischer Mappings auf den Wikipedia Artikel. Diese Mappings können ohne Änderungen am Code erstellt werden und liegen als separate XML-Dateien vor. Wegen dieser Mappings ist der MappingExtractor komplexer einzuschätzen als der InfoboxExtractor. Ursprünglich geplant, wurde dieser Extraktor jedoch **nicht** im Laufe dieser Bachelorarbeit angepasst, da er in naher Zukunft (Stand Anfang 2018) durch einen RML-Mappings basierten Extraktor ausgetauscht werden soll. Da dieser jedoch noch nicht funktionstüchtig ist und die Anpassung des MappingExtractor einen hohen Aufwand erfordert, wird dieser Extraktor erst in einem separat folgendem Projekt behandelt.

3.1 Wie kann Apache Spark die Dump-Extraktion unterstützen?

Für die Beschleunigung der Extraktion wurde zu Beginn direkt Apache Spark vorgeschlagen. Dennoch sollte vor der genauen Konzeption ein Überblick geschaffen werden, wie nützlich Apache Spark für das Extraction-Framework wirklich ist und wie gut es in den Ablauf der Dump-Extraktion passt.
Apache Sparks ursprüngliches Nutzungsfeld ist die verteilte Batchverarbeitung. Also vollständiges Verarbeiten vorliegender Datensätze auf verteilten Architekturen. Dies ist auch der Anwendungsfall in diesem Projekt. Inzwischen unterstützt Apache Spark jedoch auch Inputstreams durch SparkStreaming oder das effiziente Verarbeiten und Abfragen von strukturierten Daten durch SparkSQL. Diese Erweiterungen können gegebenenfalls für andere Teile des Extraction-Frameworks von Nutzen sein.

Betrachten wir nun Abbildung 3.1:
Diese Abbildung zeigt die möglichen Einsatzgebiete der verschiedenen Funktionen des Apache Spark Kernmoduls in der Dump-Extraktion. Die nach außen hin wichtigen Schnittstellen des Kernmoduls sind die Klasse `SparkContext` für die Dateneingabe und

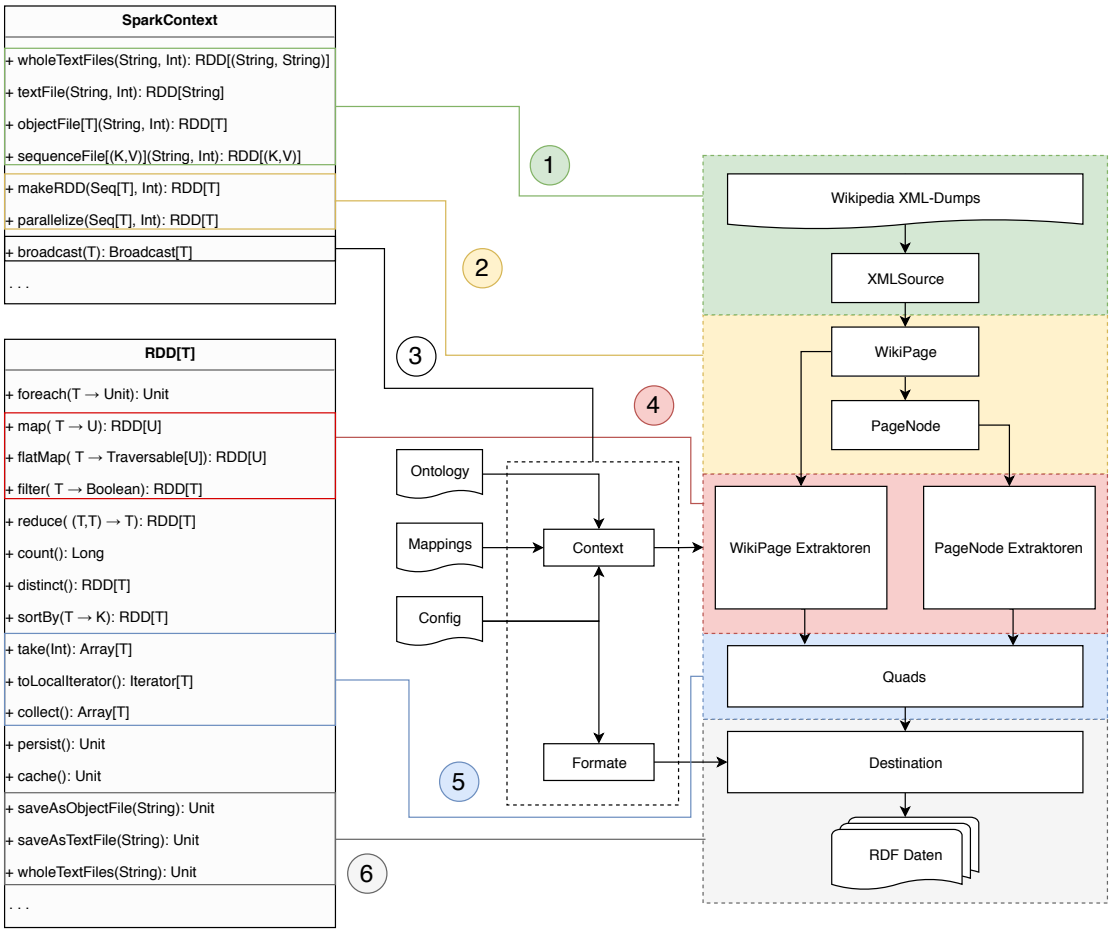

Abbildung 3.1: Möglichkeiten Apache Spark-Core im Dump-Extraktionsfluss zu nutzen

das Workermanagement und die Klasse `RDD` für die Datenverarbeitung und -ausgabe. Zur Dateneingabe können Daten können aus dem Dateisystem (siehe 3.1:1) oder dem Hauptspeicher (siehe 3.1:2) geladen werden. Dabei stehen für die Dateneingabe aus dem Dateisystem verschiedene Methoden für verschiedene Dateiformate zur Verfügung. Die Dateneingabe aus dem Hauptspeicher kann über die `makeRDD`- oder `parallelize`-Funktion geschehen. Hier wird eine Scala Kollektion partitioniert, an die Worker verteilt und somit zu einer RDD transformiert. Dies kann bei der Dump-Extraction verwendet werden, um den Strom aus WikiPage-Instanzen der XMLSource Klasse zu einer RDD zu transformieren.
Zusätzlich zur Dateneingabe bietet die Klasse SparkContext auch die Möglichkeit Objekte mit der `broadcast`-Funktion einmalig an alle Worker zu versenden. Dies kann verwendet werden, um effizient statische Daten wie das Context-Objekt an alle Worker zu versenden (siehe 3.1:3).
Die verteilten Daten werden als RDD repräsentiert. Die RDD Schnittstelle bietet eine Vielzahl an Operationen, die nahezu freie Transformationen der Daten ermöglichen. Dies ist besonders nützlich für die Extraktion, da so die Transformation WikiPage zu Quads direkt in einem Schritt durch die `flatMap`-Methode auf der RDD ausgeführt

werden kann (siehe 3.1:4). `flatMap` führt eine Funktion auf der RDD aus, die eine eindimensionale Kollektion zurück gibt. Diese Kollektion wird dann direkt aufgelöst, sodass wir eine RDD aus allen Objekten dieser Kollektionen erhalten. Die Methoden `map`, `flatMap` und `foreach` verhalten sich dabei analog zu den gleichnamigen Methoden der Scala-Kollektionen.
Die Datenausgabe kann analog zur Dateneingabe auch wieder in den Hauptspeicher (siehe 3.1:5) oder in das Dateisystem (siehe 3.1:6) ausgegeben werden. Um Elemente aus einer RDD direkt weiter im Hauptspeicher weiterverwenden zu können, müssen die Inhalte der RDD erst mit einer der Methoden `take`, `toLocalIterator` oder `collect` zum Master geschickt werden. Mit `take` nimmt man dabei nur eine bestimmte Anzahl an Elementen, mit `toLocalIterator` die einzelnen Elemente sobald sie fertig berechnet sind und mit `collect` die fertig gestellten Partitionen.
In das Dateisystem können die Daten ebenso in verschiedenen Formaten ausgegeben werden.
Die Eingabe und die Ausgabe können auch in ein Hadoop-Dateisystem ausgeführt werden. So eine Skalierung ist zwar noch nicht geplant, aber in Zukunft durchaus Möglich und durch Apache Spark einfach umzusetzen. Durch die native Unterstützung von BZip2-Komprimierung muss für die Dateneingabe/Ausgabe des Extraction-Frameworks keine Änderung an den Datensätzen vorgenommen werden.
Apache Spark bietet noch viele weitere Funktionen die für andere Module des Extraction-Frameworks nützlich sein könnten, wie zum Beispiel eine Duplikatentfernung mit `distinct`, einen flexiblen Datenfilter `filter` oder eine Sortierung mit `sortBy`. Diese werden jedoch nicht direkt in der Dump-Extraktion benötigt.

Fehlend ist eine direkte Unterstützung von strukturierten oder semi-strukturierten Daten wie zum Beispiel XML. Die Daten werden mit den Standardmethoden entweder nur Zeile-für-Zeile oder Datei für Datei gelesen. Durch Zeile-für-Zeile würden XML-Elemente so über mehrere Worker verteilt werden. Bei Datei-für-Datei müssten die Datensätze vorher noch aufgeteilt werden. Hierfür müsste bei einem Ansatz direkt am Dateisystem eine andere Methode gefunden und implementiert werden.
Bei einer Ausgabe in das Dateisystem müssten alle Funktionen der Destination-Klasse durch eine Ausgabefunktion von Apache Spark gelöst werden. Dies ist mit den Standardmethoden ebenso nicht möglich. Hierfür müsste auch eine geeignete Ausgabemethode gefunden und implementiert werden.
Aus diesen Gründen wird zunächst ein Prototyp mit dem einfachen Hauptspeicheransatz entwickelt, um den Aufwand, die voraussichtlichen Probleme und die Effizienz einschätzen zu können.

3.2 Der naive Prototyp

3.2.1 Dateneingabe

Für die Dateneingabe werden die Daten aus dem WikiPage-Stream der XMLSource Klasse in den Hauptspeicher geladen und an Apache Spark weitergegeben. Durch die `makeRDD` Methode wird die Daten parallelisiert, partitioniert und auf die Worker-Knoten verteilt.

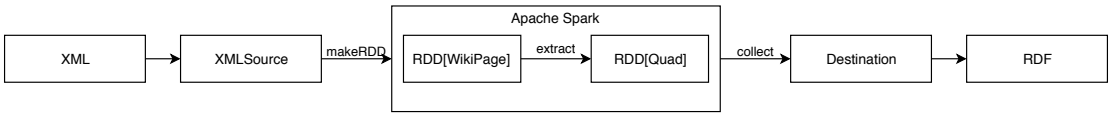

Abbildung 3.2: Datenfluss: Prototyp

Nach erster Ausführung stellt man jedoch fest, dass dieser Ansatz nicht effizient ist, da Apache Spark die Daten erst partitioniert und weitergibt, wenn die kompletten Daten im Hauptspeicher liegen. Da die Eingabedaten jedoch in komprimierter Form schon mehrere Gigabyte umfassen, kann dies schnell zum Überlauf des Hauptspeichers führen. Außerdem dauert das sequentielle Lesen und Parsen einige Zeit, während dieser Apache Spark nichts berechnet.
Für die Eingabe ist demnach der naive Ansatz ungenügend.

3.2.2 Datenverarbeitung

Für die Extraktion werden die `extract`-Methoden der Extraktoren durch Apache Spark in einer Transformation auf der WikiPage-RDD ausgeführt. Dies sorgte für Serialisierungsprobleme, die im Falle des LabelExtractor schnell behoben werden konnten. Das Context Objekt, welches von den Extractor-Klassen benötigt wird, wurde auf die absolut notwendigen Inhalte verkleinert, um die Serialisierung einfacher zu gestalten. Ein weiteres Problem welches ermittelt wurde, ist die Ineffizienz beim Versenden des Context-Objektes. Das Context-Objekt ändert sich während der Laufzeit nicht und wird trotzdem an jeden Worker-Knoten immer wieder neu geschickt. Dies muss durch einmalige Broadcasts zu Beginn der Extraktion gelöst werden.

3.2.3 Datenausgabe

Für die Datenausgabe wurden die Daten nach der Extraktion durch die `collect` Methode wieder bei dem Master gesammelt und an die Destination-Klasse zum Schreiben in das Dateisystem weitergegeben. Dabei wurde festgestellt, dass die `collect` Methode die Daten nur aufsammelt sobald eine komplette Partition fertiggestellt wurde. In dieser Zeit könnte bei dem Master schon geschrieben werden.
Deshalb wurde in einer ersten Verbesserung die Methode `collect` durch die Methode `toLocalIterator` ausgetauscht, welche alle fertigen Daten direkt an den Master senden lässt. So entsteht jedoch ein zu großer Engpass bei dem Master. Hierfür muss eine Lösung gefunden, welche den Engpass umgeht. Dies kann einfacher durch den Ansatz direkt am Dateisystem gelöst werden.

3.3 Arbeitspakete

Aus der Auswertung des Prototyps ergaben sich drei Arbeitspakete die im Laufe dieser Bachelorarbeit bearbeitet wurden:

- **Dateneingabe:**
  Es muss eine geeignete Eingabemethode für Apache Spark gefunden werden, die

XML Daten effizient lesen kann. Anschließend muss das XML analog zur XMLSource-Methode zu WikiPage-Objekten geparst werden.

- **Datenverarbeitung:**
  Das Konzept des Prototyps funktioniert, jedoch muss das Problem der Serialisierung bei komplexeren Extraktoren gelöst werden und der Context möglichst vollständig mit einem Broadcast verteilt werden.
- **Datenausgabe:**
  Für die Datenausgabe muss eine Ausgabemethode für Apache Spark gefunden werden, welche das Problem des Engpasses bei dem Master umgeht und alle Funktionen der Destination-Klasse umsetzt.

Außerdem wurden noch allgemeine Optimierungen und eine Evaluation im Hinblick auf die Anforderungen eingeplant und durchgeführt.

Kapitel 4

Implementierung

In diesem Kapitel wird die Durchführung der Arbeitspakete aus dem letzten Kapitel beschrieben. Dabei werden an bestimmten Stellen Codeabschnitte oder Beispiele präsentiert, welche die Version repräsentieren, die im Nachhinein zur Evaluierung benutzt wurden.
Die präsentierten Codeabschnitte, die den Datenfluss betreffen, beziehen sich auf die Klasse SparkExtractionJob, welche den Datenfluss der Extraktion steuert.

4.1 Dateneingabe

Das erste definierte Paket ist die Dateneingabe.
Zunächst musste eine geeignete Eingabemethode für Apache Spark gefunden werden, welche das Lesen von großen XML-Dateien effizient unterstützt. Hierfür wurden zur Betrachtung auch die Spark Erweiterungen SparkSQL und SparkStreaming hinzugezogen. SparkSQL wurde untersucht, da es das Lesen von strukturierten Daten unterstützt und eine XML-Erweiterung bietet. SparkStreaming hingegen wurde hinzugezogen, da es neben Streams aus dem Netzwerk auch lokale FileStreams unterstützt und gegebenenfalls sogar direkt den XMLSource WikiPage-Stream nutzen kann. Für beide Erweiterungen wurde jeweils ein einfacher Prototyp entwickelt und getestet. Aufgefallen ist, dass die SparkStreaming Variante langsamer ist und sich dank seines 24/7-Prinzips nicht automatisch beenden konnte, sobald die Datei fertig gelesen wurde. Da der SparkStreaming Ansatz nicht mit der Geschwindigkeit des SparkSQL-Ansatzes mithalten konnte, wurde SparkStreaming nicht weiter betrachtet.
Der SparkSQL-Ansatz war zu diesem Zeitpunkt der schnellste Leseansatz. Jedoch sind SparkSQLs DataFrames nicht so flexibel bei der Serialisierung wie die RDDs des Kernpakets, wodurch eigene Encoder für jeden Datentypen, der in den DataFrames liegen soll, geschrieben werden müssten. Um dies zu umgehen, könnte man für die anschließende Datenverarbeitung den DataFrame zu einer RDD transformieren. Dies kostet wiederum Zeit.
Außerdem besitzt die Ausführung durch SparkSQL einen Overhead in dem die Daten gelesen und für Anfragen optimiert werden. Dieser Overhead bringt in diesem Use-Case kaum Gewinn, da die Daten nur ein einziges Mal verarbeitet werden.
Also wurde noch einmal ein genauerer Blick auf das Kernpaket Apache Sparks gewor-

fen. Dabei wurde festgestellt, dass die Methode `textFile` eine speziell konfigurierte Variante der `hadoopFile`-Methode ist. Die Methoden für die Eingabe und Ausgabe in ein Hadoop-Dateisystem wurden zuvor nicht betrachtet, da zu diesem Zeitpunkt keine Hadoop-Nutzung geplant wurde und sie in der recherchierten Literatur nicht ausführlich beschrieben wurden. Diese Methoden unterstützen jedoch nicht nur das Hadoop-File-System, sondern auch lokale Dateisysteme. Diese Erkenntnis führte zur Untersuchung anderer Methoden wie `newAPIHadoopFile`, welche zusätzliche Konfigurationen bei der Dateneingabe ermöglicht. Dadurch konnte die Methode analog zu `textFile` implementiert werden, mit dem Unterschied, dass hier das Start-Tag des `<page>`-Elements als Trennzeichen dient.

```
1 val numberOfCores = sparkContext.defaultParallelism
2
3 [...]
4
5 val inputConfiguration = new org.apache.hadoop.conf.Configuration
6 inputConfiguration.set("textinputformat.record.delimiter", "<page>")
7 val wikipageXmlRDD : RDD[String] =
8 sparkContext.newAPIHadoopFile(
9     sourcePath,
10    classOf[TextInputFormat],
11    classOf[LongWritable], classOf[Text],
12    inputConfiguration
13    ).map("<page>" + _._2.toString.trim).repartition(numberOfCores)
```

Listing 4.1: Dateneingabe

Wie man in dem Codeabschnitt 4.1 sieht, wurde zunächst die Anzahl der verwendeten Kerne aus dem SparkContext bestimmt. Dieser wird später verwendet, um die Anzahl an Partitionen festzulegen. Anschließend wurde mit der Methode `newAPIHadoopFile` und einer leicht editierten Hadoop Konfiguration der Datensatz direkt in eine RDD eingelesen. Die Daten liegen zu diesem Zeitpunkt als Tupel vor, welches den Artikeltext als zweiten Wert enthält. Durch Nutzung der `map`-Methode wurde der gelesene Text aus dem Tupel extrahiert und das, durch die Trennung verloren gegangene, Start-Tag konkateniert. Anschließend wurde die RDD auf die gewünschte Anzahl an Partitionen repartitioniert. Dies ist notwendig, da die Daten sonst nur auf zwei Partitionen aufgeteilt werden. Nach dem offiziellen Tuning-Guide [1] wird empfohlen zwei bis drei Partitionen pro Worker zu nutzen, jedoch lieferte in diesem Fall die Variante mit einer Partition pro Worker bessere Ergebnisse.
Als Nächstes mussten die XML-Strings zu WikiPage-Objekten geparst werden. Dabei wurde hauptsächlich der Parser-Code aus der XMLSource-Klasse wiederverwendet und zusätzlich das erste Objekt aussortiert, da dieses nur den XML Prolog und keine WIkiseite enthält.

4.2 Datenverarbeitung

Die Datenverarbeitung besteht aus zwei Hauptproblemen: Die Serialisierung und die Redundanz bezüglich Datenversand und Objekterstellung.

[1] Tuning Guide for Apache Spark: https://spark.apache.org/docs/latest/tuning.html

4.2.1 Serialisierung

Alle Nachrichten die in Apache Spark zwischen den Master- und Worker-Knoten verschickt werden, müssen in Byte-Code umgewandelt werden. Diesen Vorgang nennt man Serialisierung. Schwierig wird die anschließenden Deserialisierung, der Vorgang, Byte-Code wieder zu einer funktionstüchtigen Objekt-Instanz zu transformieren. Java ermöglicht Serialisierung schon durch die Standardbibliotheken, aber es existiert auch das Software-Framework Kryo [2], welches damit wirbt um einiges schneller als Javas Standardimplementierung zu sein.
Die Serialisierung durch Java wurde implementiert, indem man die Klasse `Serializable` aus der Java Standardbibliothek durch Vererbung erweitert. Zu beachten ist, dass alle in der Klasse referenzierten Objekte auch serialisierbar sein müssen. Um Probleme mit der Serialisierung zu vermeiden und die Kommunikationszeiten zwischen den Knoten zu minimieren, sollten die serialisierbaren Klassen möglichst kurz gehalten werden und wenig Referenzen auf andere Objekte enthalten. Apache Spark unterstützt sowohl Javas Serialisierung als auch Kryo ohne zusätzliche Paketinstallation. Kryo kann durch Konfiguration des SparkContext Objektes verwendet werden.

In dem Fall vom Extraction-Framework müssen neben WikiPage, Quad, den Elementen aus dem Context auch die Extraktoren serialisierbar gemacht werden.

```
1 trait Extractor[-N] extends java.io.Serializable {
2     [...]
3 }
```

Listing 4.2: Serialisierung

Da die Extraktoren alle das Extractor-Interface erweitern, kann die Serializable Vererbung mit sehr geringem Aufwand wie im Codeabschnitt 4.2 implementiert werden. Als Nächstes wird sichergestellt, dass die Extraktoren nur serialisierbare Klassen referenzieren. Dabei wurde festgestellt, dass viele Extraktoren verschiedene Parser-Klassen nutzen. Diese besitzen teilweise Logger-Objekte, die zur Konsolenausgabe der Parser-Fehler dienen.

```
1 @transient private val logger = Logger.getLogger(getClass.getName)
```

Listing 4.3: Logger Handhabung

Logger-Objekte können nicht serialisiert werden. Dieses Problem umgeht man, indem man sie wie im Code-Beispiel 4.3 mit dem `@transient`-Tag versieht. Dies lässt die Worker das Logger-Objekt neu initialisieren, anstatt zu versuchen die Logger-Instanz des Masters zu verwenden. Dies beeinträchtigt die Loggerausgabe in lokaler Ausführung nicht.

Ein weiteres Serialisierungsproblem stellt das Context-Objekt dar. Elemente wie die Ontology, Language oder Redirects lassen sich sehr einfach Serialisieren, Mappings hingegen sind ein viel komplexeres Konstrukt. Die Mappings werden nur durch den MappingExtractor genutzt. Durch die Nutzung von Apache Spark, müssten sie jedoch bei jeder Extraktion an alle Worker geschickt werden. Daher wäre es aus Effizienzgründen angebracht für dieses Objekt eine eigene Lösung zu finden. Dies gilt ebenso für die Elemente CommonsSource und Disambiguations. Da diese Extraktoren aktuell noch nicht

[2] Kryo: https://github.com/EsotericSoftware/kryo

umgesetzt werden sollen, werden diese Elemente vorerst aus dem Context entfernt und erst in einem später folgendem Projekt wieder hinzugefügt. Ein weiteres entferntes Element ist die articleSource. Dieses Element war der WikiPage Stream welcher als Input für die Extraktoren diente. Da die Dateneingabe nun jedoch über Apache Spark direkt geregelt wird, muss dieses Element nicht mehr an die Extraktoren verschickt werden.

Zu Serialisierungszwecken wurden die XML-Parser und Quad-Extraktions Methoden in einer eigene serialisierbare Klasse implementiert, um Referenzen auf nicht serialisierbare Objekte minimal zu halten.

4.2.2 Redundanz

Einige Objekte und Variablen die in der Datenverarbeitung verwendet werden ändern sich kaum. Dies trifft zum Beispiel auf das `Context`-Element zu. Der Context wird zu Beginn der Extraktion mit Daten aus der Config, der Ontologiedatei, den Mappings und anderen externen Quellen gefüllt und danach nicht mehr verändert. Trotzdem würde er bei der Datenverarbeitung über die RDDs im Extraktionsschritt bei jedem WikiPage Objekt mitgeschickt.

```
1 //broadcasts
2 val broadcastedNamespaces = sparkContext.broadcast(namespaces)
3 val broadcastedOntology = sparkContext.broadcast(context.ontology)
4 val broadcastedLanguage = sparkContext.broadcast(context.language)
5 val broadcastedRedirects = sparkContext.broadcast(context.redirects)
6 val broadcastedFormats = sparkContext.broadcast(config.formats)
7 val broadcastedExtractors = sparkContext.broadcast(extractors)
```

Listing 4.4: Broadcasts

Diese Form der Redundanz lässt sich durch die Nutzung der `broadcast`-Methode des SparkContext-Objektes, wie im Codeabschnitt 4.4 dargestellt, reduzieren. Hier werden Elemente, die sich während der laufenden Extraktion nicht ändern, durch die Broadcast-Methode an alle Worker verteilt. Diese Elemente sind zum Beispiel die Inhalte des Context-Objektes oder die Liste an Extraktoren und Namespaces mit denen die Extraktion ausgeführt werden soll.

Wenn der Context nur einmal an die Worker-Knoten geschickt werden soll, darf jedoch die Extractor-Klasse erst auf dem Worker-Knoten initialisiert werden, da der Context nur bei Erstellung an die Extractor-Instanz übergeben wird. Dies führt zur wiederholten Erstellung eines gleichbleibenden Objektes.

```
1 val extractedDataRDD : RDD[Quad] = wikipageParsedRDD.mapPartitions(pages => {
2   def worker_context = new SparkExtractionContext {
3     def ontology: Ontology = broadcastedOntology.value
4     def language: Language = broadcastedLanguage.value
5     def redirects: Redirects = broadcastedRedirects.value
6   }
7   val localExtractor = CompositeParseExtractor.load(broadcastedExtractors.value,
      worker_context)
8   pages.map(page =>
9     SerializableUtils.extractQuadsFromPage(broadcastedNamespaces.value,
      localExtractor, page)
10  )
```

```
11 }).flatMap(identity)
```

Listing 4.5: Nutzung von mapPartitions, um Objekterstellung zu reduzieren

Dieses Problem wird im Code-Abschitt 4.5 gelöst. Hier wird die Nutzung von `mapPartitions` anstelle von `map` gezeigt. Die `mapPartitions`-Methode ermöglicht die Ausführung von Code einmalig pro Partition. Dies wird ausgenutzt, um die Context- und Extraktor-Objekte nur einmal pro Partition zu erstellen. Da in diesem Fall nur eine Partition pro Worker genutzt wird, werden sie sogar nur einmal pro Worker erstellt. Außerdem sieht man in 4.5 wie die durch `broadcast` verteilten Elemente in den Workern verwendet werden. Dies geschieht durch Extraktion der Werte aus den im Codeabschnitt 4.4 erstellten Broadcast-Objekten durch die `value`-Methode.

```
1  def extractQuadsFromPage(namespaces: Set[Namespace], extractor: Extractor[WikiPage
       ], page: WikiPage): Seq[Quad] = {
2    var quads = Seq[Quad]()
3    try {
4      if (namespaces.exists(_.equals(page.title.namespace))) {
5        //Extract Quads
6        quads = extractor.extract(page, page.uri)
7      }
8    }
9    catch {
10     case ex: Exception =>
11       page.addExtractionRecord(null, ex)
12       page.getExtractionRecords()
13   }
14   quads
15 }
```

Listing 4.6: Extraktion

Mit den erstellten Extraktoren kann nun die eigentliche Extraktion im Codeabschnitt 4.6 durchgeführt werden. Dabei wird wie in der originalen ExtractionJob Klasse zuerst der Namespace der WikiPage überprüft und anschließend die `extract`-Methode des Extraktors ausgeführt. Fehler werden in dem WikiPage Objekt selbst geloggt, eine Funktionalität, die aktuell noch nicht verwendet wird, da die WikiPage-Instanz nach diesem Schritt zu Quad-Objekten umgewandelt wurde.

4.3 Datenausgabe

Der Prototyp zeigte, dass die Sammlung der Daten beim Master durch die `collect`- oder `toLocalIterator`-Methoden zu einem Engpass führten. Um dies zu vermeiden muss jeder Worker-Knoten seine eigenen Dateien schreiben. Im Anschluss darauf können die Dateien dann durch ein einfaches Bash-Skript konkateniert werden. Dazu wird wiedereinmal die Teilbarkeit der BZip2-Komprimierung ausgenutzt. Um dies durchzuführen, muss jedoch zuerst eine Ausgabemethode gewählt werden, die die Aufteilung der Daten nach beliebigen Kriterien zulässt. Da die Ausgabeschnittstelle in vielen Punkten analog zur Eingabeschnittstelle läuft, kann man hier direkt die Erkenntnis nutzen, dass komplexe Einstellungen an den jeweiligen Methoden nur durch die Hadoop-Eingabe- und

Ausgabemethoden möglich sind. Daher war es naheliegend sich direkt mit der Methode `saveAsHadoopFile` zu beschäftigen. Diese unterstützt das Schreiben von mehreren Dateien, mit Unterteilung nach einem Schlüsselelement. Dabei kann ein eigenes Ausgabeformat erstellt werden, welches die weitere Funktionalitäten ermöglicht.

4.3.1 Generierung des Schlüssels

Durch die Methode `mapPartitionsWithIndex` kann zunächst der Index der jeweiligen Partition bestimmt und als Teil des Schlüsselelements verwendet werden. Somit wird erreicht, dass die Daten jeder Partition in eine Datei geschrieben werden. Da die Partitionen nie über mehrere RDDs verteilt sind, werden die Resultate nie zwischen den Knoten verschickt. Damit ist das Engpassproblem der Ausgabe gelöst.

```
1 extractedDataRDD.mapPartitionsWithIndex(
2     (partition, quads) => quads.flatMap(quad =>
3         broadcastedFormats.value.flatMap(formats =>
4             Try{(Key(quad.dataset, formats._1, partition), formats._2.render(quad).
      trim)}.toOption
5         )
6     )
7 ).saveAsHadoopFile(s"${broadcastedDir.value}/_temporary/", classOf[Key], classOf[
      String], classOf[CustomPartitionedOutputFormat], classOf[BZip2Codec])
8 concatOutputFiles(new File(s"${broadcastedDir.value}/_temporary/"), s"${lang.
      wikiCode}${config.wikiName}-$latestDate-", broadcastedFormats.value.keys)
```

Listing 4.7: Generierung des Schlüssels und Datenausgabe

Es müssen jedoch noch zwei wichtige Funktionalitäten der Destination-Klasse umgesetzt werden: die Aufteilung der Daten nach Datensatz und die Umwandlung der Daten in die gewünschten Ausgabeformate. Die Ergebnisse der Extraktion sind Quad-Objekte, welche den Zieldatensatz enthalten. Fügt man diesen zu dem Schlüsselelement hinzu, kann die Aufteilung nach Datensatz gewährleistet werden. Das Problem hierbei ist die anschließende Konkatenation der Dateien, welche dadurch komplizierter wird.

```
1 class CustomPartitionedOutputFormat extends MultipleTextOutputFormat[Any, Any] {
2   override def generateActualKey(key: Any, value: Any): Any =
3     NullWritable.get()
4
5   override def generateFileNameForKeyValue(key: Any, value: Any, name: String):
      String = {
6     val k = key.asInstanceOf[Key]
7     val format = k.format.split(".bz2")(0)
8     s"${k.dataset}/${k.partition}.$format"
9   }
10 }
11
12 case class Key(dataset: String, format: String, partition: Int)
```

Listing 4.8: Ausgabeformat und Schlüssel-Hilfsklasse

Die konkrete Aufteilung in eigene Dateien wurde durch einen eigenen Hadoop-Ausgabetyp, welcher im Codeabschnitt 4.8 implementiert wurde, umgesetzt. Dieser kann durch die Methode `generateFileNameForKeyValue` den Zielpfad der Ausgabe durch das Schlüsselelement generieren. Sorgt man hier dafür, dass der Datensatz als Ordnername ver-

wendet wird, werden die Daten sauber nach Datensatz in verschiedene Ordner unterteilt und so einfacher durch ein Bash-Skript kombinierbar. Das Bash-Skript muss nun nur noch auf allen Datensatzordnern ausgeführt werden.
Eine weitere Funktionalität der Desitination Klasse ist das Schreiben der Daten in den gewünschten Formaten. Dies kann durch eine Transformation auf der Quad-RDD gelöst werden, welche für jedes Quad für jedes Format einen eigenene Schlüssel erstellt und das Quad direkt in das gewünschte Format umwandelt. Dabei wurde der Schlüssel, um die Formatendung erweitert. Anschließend war nur noch eine weitere Anpassung des Ausgabeformats nötig.
Die Dateien werden von Apache Spark in einem eigenen Ordner erstellt, welcher nach vollständiger Extraktion und Konkatenation der Dateien wieder gelöscht wird um direkt weiteren Betrieb zu ermöglichen.

4.4 Wahrung der originalen Funktionalität

Eine letzte Anforderung an dieses Projekt war die Wahrung der ursprünglichen Extraktionsfunktion, sodass diese noch ausführbar und voll funktionstüchtig ist. Deshalb wurden im Laufe dieser Implementierung möglichst wenig Änderungen an bestehendem Code durchgeführt. Alle Klassen, die stark verändert werden mussten wurden als Kopie unter abgewandeltem Namen implementiert. Änderungen, die an Originaldateien durchgeführt wurden waren zum Beispiel die Herstellung der Serialisierbarkeit oder Erweiterungen, um zusätzliche Funktionen, die die originale Extraktionsausführung nicht beeinträchtigen.

```
1 $ ~/extraction-framework/dump> ../run extraction extraction.default.properties
2 $ ~/extraction-framework/dump> ../run sparkextraction extraction.spark.properties
```

Listing 4.9: Ausführung der Dump-Extraction

Die Dump-Extraktion wird wie in Abbildung 4.9 durch Ausführung des `Skriptes` mit dem Argument `extraction` gestartet. Die Apache Spark-gestützte Variante ist durch starten des `run`-Skriptes mit dem Argument `sparkextraction` möglich. Dabei wird eine eigene Konfiguration mitgegeben, welche die Konfiguration des Apache Spark Masters ermöglicht.

Kapitel 5

Ergebnisse

Dieses Kapitel beschäftigt sich mit den Ergebnissen der im letzten Kapitel vorgestellten Implementierung. Es werden Laufzeitvergleiche durchgeführt, das Verhalten bei Nutzung verschiedener Ressourcen beobachtet und die Datenvollständigkeit überprüft. Anschließend wird ein Überblick über den Fortschritt der Gesamtimplementierung gegeben und der Aufwand der noch fehlednen Funktionalitäten eingeschätzt.

5.1 Vollständigkeit der Daten

Nach Anforderung 3 sollen die extrahierten Daten der Apache Spark Implementierung mit denen der ursprünglichen Extraktion verglichen werden und auf Vollständigkeit überprüft werden. Dabei ist die Reihenfolge der extrahierten Daten unwichtig, das wichtige Kriterium ist die Vollständigkeit.
Um die Datenvollständigkeit zu überprüfen, wurden verschiedene Tests auf verschiede-

Extraktor	Datensatz	Sprache	Artikel	Master	Spark
Label	labels	de	8	8	8
Label	labels	en	8	8	8
Infobox	infobox-properties	de	8	51	51
Infobox	infobox-properties	en	8	6	6
Infobox	infobox-property-def..	de	8	86	86
Infobox	infobox-property-def..	en	8	12	12
Label	labels	de	10000	9999	9999
Label	labels	en	10000	10000	10000
Infobox	infobox-properties	de	10000	99540	99540
Infobox	infobox-properties	en	10000	97866	97866
Infobox	infobox-property-def..	de	10000	11022	11022
Infobox	infobox-property-def..	en	10000	5032	5032
Infobox	infobox-properties	de	3.911.781	20.470.755	20.470.755
Infobox	infobox-properties	en	17.008.269	61.267.688	61.267.688

Abbildung 5.1: Datenvollständigkeit

nen Datensätzen ausgeführt. Die Tests umfassen verschiedene Datengrößen, die Datensatzsprachen Deutsch und Englisch und die Extraktoren LabelExtractor und InfoboxExtractor. Die extrahierten Tripel wurden anschließend mit `$ sort -u` sortiert, gezählt und bei Stichprobenartig durch `$ comm -3` verglichen. Dadurch wurde sichergestellt, dass die extrahierten Daten vollständig sind.
Die Ergebnisse dieses Tests werden in Tabelle 5.1 dargestellt. Zu Beginn wurden kleine Testdatensätze für die deutsche und englische Sprache erstellt, die jeweils acht Artikel umfassen. Diese Zahl wurde gewählt um mindestens 8 Partitionen erstellen zu können und die Parallelisierung zu testen. Auf diesen Testdatensätzen wurde das originale Extraction-Framework ausgeführt und die extrahierten Daten als Referenzergebnisse gesichert. Dann wurde die Apache Spark-Integration auf den Testdatensätzen ausgeführt und die Ergebnisse mit den Referenzdaten verglichen. Die Ergebnisse davon befinden sich in den ersten sechs Zeilen der Tabelle in Tabelle 5.1. Die Spalte *Master* enthält dabei die Tripelanzahl des alten Extraction-Frameworks abzüglich der Kommentare und die Spalte *Spark*, die Tripelanzahl der Apache Spark Implementierung. Bei diesen Tests traten keine fehlenden Daten auf. Dennoch sei dazu gesagt, dass die Tripelanzahl bei einer Sortierung ohne die Option `-u` nicht übereinstimmt. Diese Option filtert alle duplizierten Zeilen aus dem Datensatz. Doppelte Tripel treten bei den extrahierten Daten der Apache Spark-Integration auf, da die Destination-Klasse eine eigene Implementierung eines Doppelungsfilters besitzt, die nicht ohne großen Zeitaufwand in Apache Spark umzusetzen war. Dies ist jedoch kein großer Makel, da dies die Datennutzung nicht beeinträchtigt. Werden die Daten zum Beispiel in einen Tripel-Store geladen, werden die Doppelungen dort automatisch entfernt.
Im Anschluss wurden für beide Sprachen Datensätze erstellt, die aus 10.000 Artikeln der jeweiligen Dumps bestehen. Auf diesen wurden wieder beide Versionen der Dump-Extraktion ausgeführt und verglichen. Diese Tests wurden für erste Laufzeitvergleiche durchgeführt. Auch hier gab es bis auf Doppelungen keine Vorkommnisse. Schließlich wurden die Extraktoren auf den vollständigen Datensätzen der beiden Sprachen ausgeführt und die Ergebnisse verglichen. Dies sind die letzten beiden Zeilen in Tabelle 5.1. Dieser Test wurde ausgeführt, um den Härtefall zu testen: hohe Anzahl Artikel, hohe Anzahl extrahierte Tripel und Nutzung von 64 Kernen. Aber auch hier ist der einzige Unterschied die doppelten Tripel. Die Anzahl der doppelten Tripel ist jedoch mit 133 von über 20 Millionen so gering, dass dieser zu vernachlässigen ist.
Anforderung 3 wurde somit erfüllt.

5.2 Laufzeit

Auch die Laufzeit wurde für beide Versionen der Dump-Extraktion auf den Sprachen Englisch und Deutsch, auf verschiedenen Datensatzgrößen und auf den Extraktoren LabelExtractor und InfoboxExtractor getestet. Dazu kommen noch Tests über die Veränderung durch Nutzung verschiedener Anzahlen von Prozessorkernen. Es wurden Tests auf Verschiedenen Kernanzahlen zwischen 4 und 64 Kernen auf dem DBpedia Extraction-Server (siehe: Tabelle 5.5) ausgeführt. Vier Kerne wurde als minimale Kernanzahl für die Tests ausgewählt, da das Extraction-Framework auf keiner Maschine ausgeführt werden wird die weniger Kerne besitzt. Außerdem sind die Zeitgewinne bei Nutzung von weniger als Kernen so gering, dass sich eine Apache Spark Implementierung nicht lohnt. 64

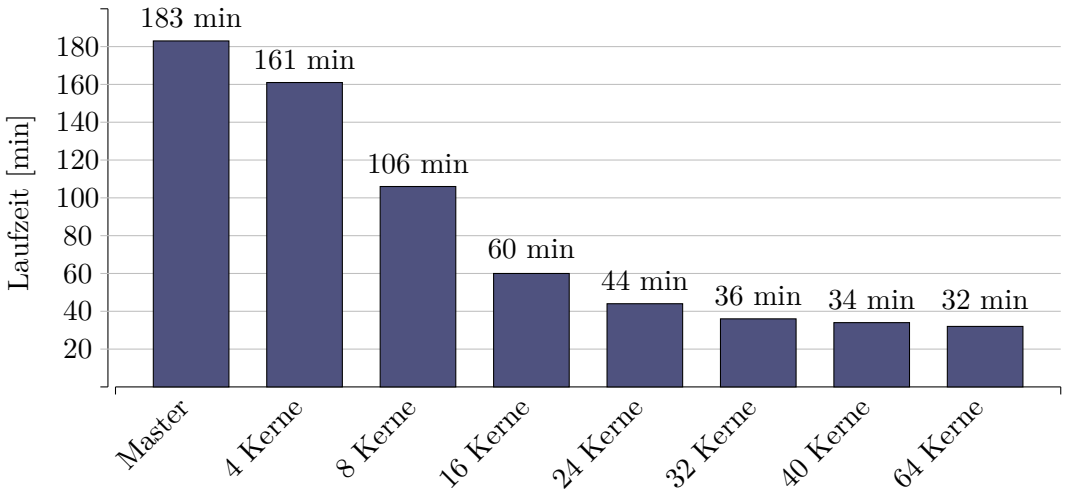

Abbildung 5.2: InfoboxExtractor, vollständiger deutscher Datensatz 2016-10

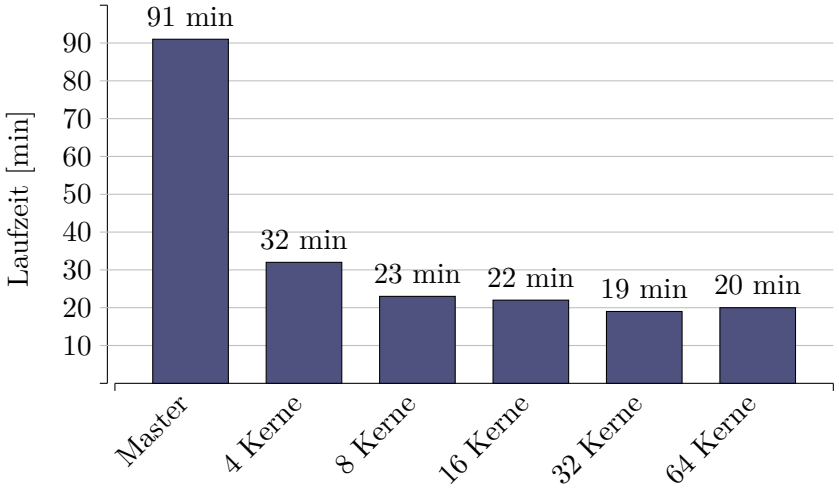

Abbildung 5.3: LabelExtractor, vollständiger deutscher Datensatz 2016-10

Kerne wurden als maximale Anzahl in den Tests verwendet, da so die maximale Anzahl an Kernen des Extraction-Servers genutzt wird.
Abbildung 5.2 zeigt die Laufzeiten eines Tests des InfoboxExtractors auf dem gesamten deutschen Datensatz der 2016-10 Extraktion. Der erste Balken stellt dabei die Vergleichszeit dar: Die Laufzeit der alten Dump-Extraktion, die 183 min beträgt. Die anderen Balken beschreiben die Laufzeiten der Apache Spark gestützten Implementierung auf verschiedenen Kernanzahlen. Dabei sieht man, dass sich der Zeitgewinn ab 16 Kernen immer weiter abnimmt und sich die Laufzeit bei etwa 33 min einpendelt. Dies entspricht einer Senkung der Laufzeit um etwa 80%. Das Sinken des Zeitgewinns pro zusätzlichem Kern liegt an dem steigenden Kommunikationsaufwand zwischen den Prozessoren und der begrenzten Lese- und Schreibgeschwindigkeit der Festplatten. Im Vergleich dazu

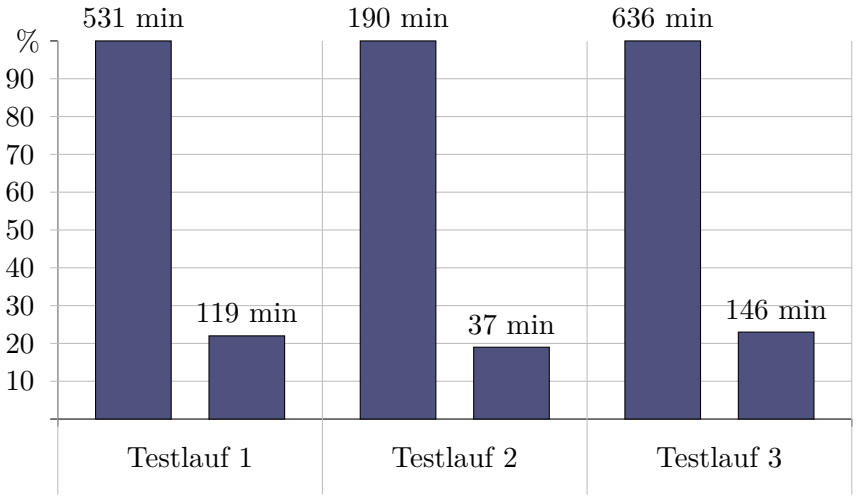

Abbildung 5.4: Weitere Vergleiche

Komponente	Spezifikation
Prozessor	4x AMD Opteron™ 6376, 16x2,3 GHz (3,2 GHz Turbo Mode)
Hauptspeicher	256 GB
Festplattenspeicher	2,5 TB SSD

Abbildung 5.5: Hardware: Extraction-Server akswnc7

steht in Abbildung 5.3 der Laufzeitvergleich des LabelExtractors, welcher sich schon nach Nutzung von 8 Kernen nicht mehr stark verändert. Dieser pendelt sich bei etwa 21 min ein, was einer Verringerung der Laufzeit von etwa 75% entspricht. Die Unterschiede zu dem InfoboxExtractor Test lassen sich durch den Komplexitätsunterschied der beiden Extraktoren erklären. Der LabelExtractor extrahiert nur den Artikeltitel, der InfoboxExtractor führt hingegen einige Parser und andere aufwendige Operationen durch. Durch die höhere Bearbeitungszeit pro Seite profitiert der InfoboxExtractor mehr von einer erhöhten Kernanzahl als der LabelExtractor.
Es wurden noch drei weitere, aufwendigere Vergleiche durchgeführt um andere Funktionen oder Härtefälle zu testen. Diese Ergebnisse werden in Abbildung 5.4 dargestellt und sind hier als direkte Vergleiche zwischen ursprünglichen Dump-Extraktion und der Apache Spark gestützten Version zu sehen. Dabei stellt die Höhe diesmal der prozentualen Laufzeitvergleich zur Ausführung des alten Extraction-Frameworks auf den gleichen Eingabedaten. So erhält man eine direkte Veranschaulichung des Zeitgewinns. Testlauf 1 zeigt dabei die Dump-Extraktion des InfoboxExtractors auf dem kompletten englischen 2016-10 Datensatz. Hierbei wurden 64 Kerne genutzt, um zu sehen wie viel Gewinn maximal erzielt werden kann. Dabei ist eine Verringerung der Laufzeit von etwa 78% zu vermerken. Testlauf 2 zeigt hier die Ausführung von zwei Extraktoren in einer Dump-Extraktion. Dabei wurden der LabelExtractor und der InfoboxExtractor mit 32 Kernen ausgeführt. Dabei wurde ein Zeitgewinn von etwa 80% erzielt. Interessant ist dieser Test im Vergleich zu der Ausführung des InfoboxExtractors alleine. Dabei war der Infobox-

Extractor alleine nur etwa eine Minute schneller als die kombinierte Extraktion. Daraus lässt sich schließen, dass ein großer Teil der verbleibenden Laufzeit durch Kommunikation, Lesen und Schreiben, anstatt durch die Datenverarbeitung, entsteht. Der letzte Test war die Ausführung auf dem aktuellen englischen 2018-04 Datensatz der englischen Wikipedia unter Benutzung aller wichtigen schon funktionierenden Extraktoren. Dies soll einen ersten Ausblick darauf geben, was für Zeitgewinne später in der regelmäßigen Ausführung zu erwarten sind. Auch hier wurde eine 78% kürzere Laufzeit erreicht. Anforderungen 5 und 6 wurde somit auch erreicht.

5.3 Schätzung des Restaufwands

Die durch Apache Spark gestützte Implementierung der Dump-Extraktion ist zwar schon mit vielen Extraktoren funktionstüchtig und bringt vielversprechende Ergebnisse, bietet jedoch noch einige Baustellen:
Die erste Baustelle sind die verbleibenden Extraktoren. Alle Extraktoren die maximal die Context-Elemente des InfoboxExtractors benötigen und keine zusätzlichen Klassen referenzieren müssen nicht weiter angepasst werden. Eine Ausnahme dafür sind Extraktoren die einen eigenen Logger verwenden, was jedoch durch das Hinzufügen des `transient`-Tags gelöst wird und deshalb keinen nennenswerten Aufwand verursacht. Um alle Extraktoren unterstützen zu können, müssen alle weiteren Elemente des Context-Objektes serialisierbar gemacht werden. Bei dem Element `disambiguations`, welches von dem dazu gehörigen DisambiguationExtractor zur Auflösung sprachlicher Mehrdeutigkeiten genutzt wird, erzeugt dieser Prozess nur geringfügigen Aufwand. Einfache Registrierung der Klasse bei Kryo oder alternativ Vererbung der Serializable Klasse, bei Nutzung Javas Serialisierung, sollten ausreichen.
Einen größeren Aufwand werden der MappingExtractor und der ImageExtractor verursachen. Der MappingExtractor benötigt die `mappings` und die `mappingPageSource`. Die mappingPageSource ist als Kollektion von WikiPage Instanzen einfach zu serialisieren, nur könnte sie bei einer gewissen Anzahl an Elementen größere Verzögerungen beim Verschicken der Context Elemente an die Worker verursachen. Das mappings Element ist hier das eigentliche Problem. Es besteht aus einer Liste an Mappings und einer HashMap von TemplateNode-Extraktoren. Diese werden durch ihren lazy Ansatz erst erstellt und dabei aus externen Dateien geladen, wenn der MappingExtractor benutzt werden soll. Dies macht die Serialisierung und den Broadcast an die Worker schwieriger. Wenn alle notwendigen Daten vor der Extraktion an die Worker gesendet werden, müssen die externen Mapping Dateien alle vor der Extraktion gelesen werden. Der MappingExtractor sollte bei der Extraktion gesondert behandelt werden, um unnötigen Versand von großen Datenmengen an die Worker zu vermeiden.
Der ImageExtractor benötigt das `commonsSource`-Element des Context, um das Copyright der extrahierten Bilder zu überprüfen. Das commonsSource Element ist ein Eingabestrom, welcher die Artikel des Wikipedia Commons Datensatzes enthält. Dieser Datensatz ist sehr groß und sollte daher nicht zusätzlich an die Worker geschickt werden. Somit wird auch hier eine Sonderbehandlung benötigt.
Die zweite Baustelle sind die anderen Arten der Extraktion, die in dieser Arbeit nicht behandelt wurden. Am ähnlichsten wären dabei die JsonNode-Extraktoren, welche auf den Wikidata Datensätzen arbeiten oder der NifExtractor welcher NIF-Daten aus den

Baustelle	Aufwand
DisambiguationExtractor	gering
ImageExtractor	mittel
MappingExtractor	mittel
NifExtraktor	hoch
JsonNode Extraktoren	mittel
Skripte	einzeln zu bewerten
Server	nicht lohnend
Live	nicht lohnend

Abbildung 5.6: Zusammenfassung: Restaufwand

WikiArtikeln extrahiert. Die JsonNode-Extraktoren benötigen durch das Eingabeformat JSON anstelle des XML eine komplett eigene Implementierung unter Nutzung Apache Sparks. Hierfür müssen die Eingabemethoden Apache Sparks ein weiteres Mal untersucht werden, um die beste Methode zum Lesen großer JSON-Datein auszuwählen. Die Serialisierung sollte dafür hier keine großen Probleme schaffen, da die JsonNode-Extraktoren keine besonderen Context-Elemente benötigen oder komplexe externe Klassen referenzieren.
Der NifExtraktor benötigt eine eigene Art von Context und nutzt die MediaWikiConnector Klasse, welche `OperatingSystemMXBean` von Java nutzt um Informationen über das System, auf dem die JVM läuft aubzurufen. Dies wird Probleme bei der Serialisierung hervorrufen. Durch den speziellen Context benötigt der NifExtraktor eine gesonderte Behandlung bei der Extraktion.
Die dritte Baustelle sind die restlichen Funktionen des Extraction-Frameworks. Hier muss für jeden Funktion gesondert bewertet werden, ob eine Apache Spark Implementierung nützlich ist. Bei einigen Skripten wie zum Beispiel dem neuen PersondataExtractor Skript wird dies der Fall sein, jedoch findet sich hier kein allgemeiner Prozess für alle Skripte der unter Nutzung Apache Sparks implementiert werden könnte. So müsste entweder ein allgemeiner Wrapper-Prozess geschaffen werden oder Apache Spark in die einzelnen Skripte manuell integriert werden. Weniger sinnvoll ist die Integration Apache Sparks in den Serveraspekt des Extraction-Frameworks. Hier werden nur einzelne Wikiseiten extrahiert was kaum durch Parallelisierung profitiert. Genauso wenig profitiert die Live-Extraktion, da diese nicht durch die Rechenzeit, sondern durch die Antwortzeit der Wikipedia API begrenzt wird. Diese Aufwandseinschätzung wird in Tabelle 5.6 noch einmal zusammengefasst.

Kapitel 6

Fazit

6.1 Überprüfung der Anforderungen:

Bevor ein Fazit gegeben werden kann, müssen die Anforderungen aus Kapitel 3 auf Erfüllung geprüft werden:

- **1:** Die Dump-Extraktion wird nun durch Apache Spark ausgeführt. Anforderung erfüllt.
- **2:** Der LabelExtractor und der InfoboxExtractor sind funktionstüchtig und wurden ausführlich getestet. Anforderung erfüllt.
- **3:** Die extrahierten Daten wurden auf Vollständigkeit getestet. Es sind jedoch Duplikate vorhanden. Anforderung erfüllt.
- **4:** Extraktoren, Ausgabeformate, Sprachen und Zielordner sind weiterhin frei konfigurierbar. Andere Konfigurationsmöglichkeiten wurden nicht beeinflusst. Anforderung erfüllt.
- **5:** Es werden nun eine frei konfigurierbare Anzahl an Prozessorkernen des Extraction-Servers genutzt. Dabei sind die Prozessoren bei einer Nutzung von 32 Kernen permanent über 75% ausgelastet. Anforderung erfüllt.
- **6:** Bei den Tests wurde eine Verringerung der Laufzeit von durchschnittlich über 75% festgestellt. Auch bei großen Datensätzen und vielen Extraktoren bleibt der Zeitgewinn groß. Anforderung erfüllt.
- **7:** Änderungen an Klassen des Extraction-Frameworks wurden so gering wie möglich gehalten und sind größtenteils erweiternder Natur. Apache Spark gestützte Dump-Extraktion lässt sich als alternativen zur ursprünglichen Dump-Extraktion ausführen. Ausführbarkeit ursprünglicher Dump-Extraktion wurde getestet. Anforderung erfüllt.
- **8:** Aufwand der restlichen Implementierung und Nutzen Apache Sparks für andere Teile des Extraction-Frameworks wurden ausführlich behandelt. Anforderung erfüllt.

6.2 Zusammenfassung der Resultate

Die Integration Apache Sparks in den Prozess der Dump-Extraktion bringt vielversprechende Ergebnisse und erfüllt alle definierten Anfragen. Im Durchschnitt wurde eine Verringerung der Laufzeit von über 75% durch die Nutzung von 32 Kernen erreicht, wobei die Kerne dauerhaft ausgelastet waren. Somit werden die Ressourcen des Extraction-Servers besser genutzt und regelmäßige Extraktionen auf den neuesten Wikipedia Dumps ermöglicht. Dabei gehen keine Daten verloren. Es muss jedoch darauf geachtet werden, dass die Daten nun Duplikate enthalten können. Die Flexibilität des Extraktion-Frameworks blieb erhalten und ein großer Teil der Extraktoren ist schon funktionstüchtig.
Die Aspekte der komponentenbasierten Software-Entwicklung wurden bei der Implementierung in grundlegenden Ansätzen durchgeführt. Kapitel 3 beschäftigte sich mit dem Finden von Gemeinsamkeiten der beiden Software-Komponenten und möglichen Schnittstellen. Kapitel 4 beschäftigte sich hingegen mit dem Füllen der Lücken zwischen diesen Schnittstellen und der richtigen Konfiguration und Nutzung der Teilkomponenten Apache Sparks.

6.3 Weiterführende Arbeiten

Als weiterführende Arbeit ist aktuell die Fertigstellung der Apache Spark Integration für den DisambiguationExtractor, ImageExtractor und MappingExtraktor geplant. Außerdem müssen noch weitere Möglichkeiten der Optimierung untersucht und (falls vielversprechend) durchgeführt werden. Das Projekt sollte außerdem auf einem Server aufgesetzt und regelmäßig, im besten Falle automatisiert, auf den aktuellsten Wikipedia Dumps ausgeführt werden.

Quellenverzeichnis

[1] Helmut Balzert. *Lehrbuch der Software-Technik. Bd. 1, Software-Entwicklung*. 2. Aufl. Heidelberg: Spektrum Akad. Verlag, 2001 (siehe S. 4).

[2] Frank Griffel. *Componentware : Konzepte und Techniken eines Softwareparadigmas*. 1. Aufl. Heidelberg: dPunkt Verlag, 1998 (siehe S. 4).

[3] Christian Robra. *Modellierung komponentenbasierter Software-Architekturen*. Saarbrücken: VDM Verlag Dr. Müller, 2007 (siehe S. 4).

[4] Wolfgang Pree. *Design-Patterns for Object-Oriented Software Development*. Workingham: Addison-Wesley, 1995 (siehe S. 4).

[5] George Heinemann T. *Component-Based Software Engineering - Putting the Pieces Together*. 1. Aufl. Addison-Wesley, 2001 (siehe S. 5).

[6] Mathias Geisler. *Semantic Web - schnell + kompakt*. entwickler.press, 2009 (siehe S. 6, 7).

[7] Rim Bray, Jean Paoli, C.M Sperberg-McQueen, Eve Maler und François Yergeau. *Extensible Markup Language (XML) 1.0*. 5. Aufl. W3C Recommendation 26 November 2008. URL: https://www.w3.org/TR/xml/ (siehe S. 6).

[8] Helmut Vonhoegen. *Einstieg in XML - Grundlagen, Praxis, Referenz*. 5. Aufl. Galileo Computing, 2009 (siehe S. 6).

[9] Guus Schreiber und Yves Raimond. *RDF 1.1 Primer*. W3C Working Group Note 24 June 2014. URL: https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/ (siehe S. 6).

[10] David Beckett, Gavin Carothers und Andy Seaborne. *RDF 1.1 N-Triples*. W3C Recommendation 25 February 2014. URL: https://www.w3.org/TR/n-triples/ (siehe S. 7).

[11] David Becket, Tim Berners-Lee, Eric Prud'hommeaux und Gavin Carothers. *RDF 1.1 Turtle*. W3C Recommendation 25 February 2014. URL: https://www.w3.org/TR/turtle/ (siehe S. 7).

[12] Jeff Gilchrist. *Parallel Data Compression with BZIP2*. Conference: Parallel, Distributed Computing und Systems, Januar 2004. URL: http://professor.unisinos.br/linds/teoinfo/parallel_bzip2.pdf (siehe S. 7).

[13] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer und Christian Bizer. *DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia*. Semantic Web 1 (2012) 1–5, IOS Press. URL: http://svn.aksw.org/papers/2013/SWJ_DBpedia/public.pdf (siehe S. 8).

[14] Pablo N. Mendes, Max Jakob, Andrés García-Silva und Christian Bizer. *DBpedia Spotlight: Shedding Light on the Web of Documents*. 7th International Conference on Semantic Systems (I-Semantics 2011). Graz, Austria, September 2011. URL: https://www.dbpedia-spotlight.org/docs/spotlight.pdf (siehe S. 9).

[15] Martin Odersky. *Programming Scala - Second Edition*. 2. Aufl. California: Artima Press, 2010 (siehe S. 9).

[16] Holden Karau und Andy Konwinski. *Learning Spark - Lightning-Fast Data Analysis*. 1. Aufl. Sebastopol: O'Reilly Media Inc., 2015 (siehe S. 13).

[17] Matei Zaharia, Reynold S. Xin, Patrick Wendel, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzales, Scott Shenker und Ion Stoica. *Apache Spark: A Unified Engine For Big Data Processing*. Communications of the ACM, November 2016, Vol. 59 No. 11, Pages 56-65. URL: https://dl.acm.org/citation.cfm?id=2934664 (siehe S. 13).

Anhang A

Abbildungsverzeichnis

A.1 Grafiken

A.2 Code-Abschnitte

A.3 Tabellen