Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kahlbrandt
Zweitgutachter: Prof. Dr. Zukunft

Eingereicht am: 1. Januar 2345

Daniel Kirchner

Thema der Arbeit
Skalierbare Datenanalyse mit Apache Spark

Stichworte
Schlüsselwort 1, Schlüsselwort 2

Kurzzusammenfassung
Dieses Dokument …
Daniel Kirchner

Title of the paper
Scalable Data Analysis with Apache Spark

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

1.1 Motivation

Die Entwicklung und Verbesserung von Frameworks zur Verarbeitung großer Datenmengen ist zur Zeit hochaktuell und sehr im Fokus von Medien und Unternehmen [VERWEIS]. Verschiedene Programme und Paradigmen konkurrieren um die schnellste, bequemste und stabilste Art großen Datenmengen einen geschäftsfördenden Nutzen abzuringen.

Mit dem Begriff „große Datenmengen“ oder „Big Data“ werden in dieser Arbeit solche Datenmengen zusammengefasst, die die Kriterien Volume, Velocity, Variety[1] erfüllen oder „Datenmengen, die nicht mehr unter Auflage bestimmter SLAs auf einzelnen Maschinen verarbeitet werden können“ [VERWEIS, Hadoop/Yarn Entwickler].

Als Unternehmen, das früh mit solchen Datenmengen konfrontiert war implementierte Google das Map-Reduce Paradigma [[DG04]] als Framework zur Ausnutzung vieler kostengünstiger Rechner für verschiedene Aufgaben (u.a. Indizierung von Webseiten und PageRank [[Pag01]]).

In Folge der Veröffentlichung dieser Idee im Jahr 2004 wurde Map-Reduce in Form der OpenSource Implementation Hadoop (gemeinsam mit einer Implementation des Google File Systems GFS, u.a.) [[SG03]] zum de-facto Standard für Big-Data-Analyseaufgaben [VERWEIS?].

Reines Map-Reduce (nach Art von Hadoop) als Programmierparadigma zur Verarbeitung großer Datenmengen zeigt jedoch in vielen Anwendungsfällen Schwächen:

- Daten, die in hoher Frequenz entstehen und schnell verarbeitet werden sollen erfordern häufiges Neustarten von Map-Reduce-Jobs. Die Folge ist kostspieliger Overhead durch Verwaltung/Scheduling der Jobs und gegebenenfalls wiederholtem Einlesen von Daten.

[1] [Lan01].

- Algorithmen die während ihrer Ausführung iterativ Zwischenergebnisse erzeugen und auf vorherige angewiesen sind (häufig bei Maschinenlernalgorithmen) können nur durch persistentes Speichern der Daten und wiederholtes Einlesen zwischen allen Iterationsschritten implementiert werden.
- Anfragen an ein solches Map-Reduce-System erfolgen imperativ in Form von kleinen Programmen. Dieses Verfahren ist offensichtlich nicht so intuitiv und leicht erlernbar wie deklarative Abfragesprachen klassischer Datenbanken (z.B. SQL).

In der Folge dieser Probleme entstanden viele Ansätze dieses Paradigma zu ersetzen, zu ergänzen oder durch übergeordnete Ebenen und High-Level-APIs zu vereinfachen [[SR14]].

Eine der Alternativen zu der Map-Reduce-Komponente in Hadoop ist die „general engine for large-scale data processing“ Apache Spark.

Ein Indiz für das steigende Interesse an diesem Produkt liefert unter anderem ein Vergleich des Interesses an Hadoop und Spark auf Google:

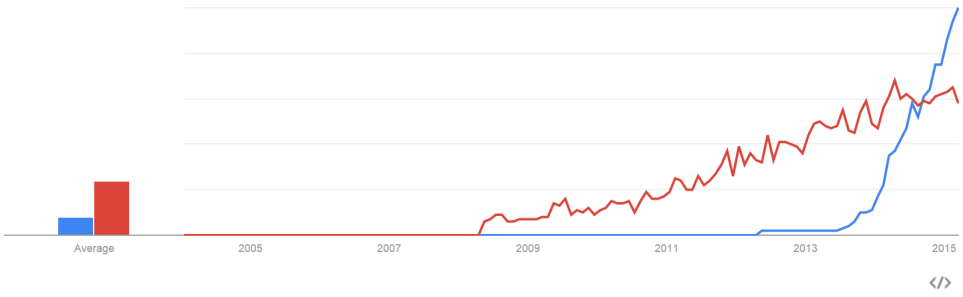


Abbildung 1.1: Vergleich der Suchanfragen zu Spark und Hadoop, Stand 24.03.2014 [[Goo]]

1.2 Kontextabgrenzung

Das Ziel dieser Arbeit ist es einen Einblick in die grundlegenden Konzepte und Anwendungsmöglichkeiten von Apache Spark zu vermitteln.

Für ein tieferes Verständnis werden zwei Anwendungsfälle untersucht und deren Lösung detailiert dokumentiert und bewertet. Hierbei kommt Apache Spark Version 1.3.0 zum Einsatz.

Nur am Rande wird betrachtet:

- Der Vergleich mit ähnlichen Produkten
- Die Empirische Messung des Skalierungsverhaltens
- Konkrete Hinweise zu Installation und Betrieb

2 Vorstellung von Apache Spark

Aus Sicht eines Nutzers ist Apache Spark eine API zum Zugriff auf Daten und deren Verarbeitung.

Diese API (wahlweise für die Sprachen Scala, Java und Python verfügbar), kann im einfachsten Fall über eine eigene Spark Konsole mit REPL-Pattern [[MH99]] verwendet werden.
Die Zählung von Wortvorkommen in einem Text - das „Hello World“ der Big Data Szene - lässt sich dort mit zwei Befehlen realisieren:

```
$ ./spark-shell
[...]
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/
Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_75)
Type in expressions to have them evaluated.
[...]
scala> val text = sc.textFile("../Heinrich Heine - Der Ex-Lebendige")
[...]
scala> :paste
text.flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _)
.collect()
[...]
res0: Array[(String, Int)] = Array((Tyrann,,1), (im,2), (Doch,1) ...)
```

Listing 2.1: Word Count in der Spark Konsole

Aus Sicht eines Administrators ist Apache Spark eine Applikation auf einem Rechencluster, die sich in der Anwendungsschicht befindet und charakteristische Anforderungen insbesondere

an Lokalität des Storages und die Netzwerkperformance stellt.

Was das konkret bedeutet, welche Mechanismen und Konzepte dahinterstehen und in welchem Ökosystem von Anwendungen sich Apache Spark bewegt wird im nächsten Abschnitt beleuchtet.

2.1 Überblick

— Was ist es, wofür ist es, wie funktioniert es, (JVM, Scala) —

2.2 Kernkonzepte

— Warum ist Spark schnell, ausfallsicher, secure (und wo vielleicht nicht)? — — Annahme: Durchsatzfaktoren RAM, Netzwerk, Festplatte —

2.2.1 Nutzung von Arbeitsspeicher

2.2.2 Nutzung von persistentem Speicher

2.2.3 Nutzung von CPUs

2.2.4 Scheduling/Shuffling

2.2.5 Kern-API

2.3 Standardbibliotheken

— Warum ist Spark so einfach (und wo vielleicht nicht)? —
Die vier Standardbibliotheken erweitern die Kern-API für bestimmte, häufig genutzte Aufgaben aus Bereichen der Datenanalyse.

Die bedienten Bereiche sind

- Deklaratives Abfragen und On-the-Fly-Transformationen (*Spark SQL*)
- Maschinenlernverfahren (*MLlib*)

- Echtzeitbehandlung von eingehenden Daten (*Streaming*)
- Operationen auf Graph-artigen Strukturen (*GraphX*)

2.3.1 Dataframes/Spark SQL

2.3.2 MLlib

2.3.3 Streaming

2.3.4 GraphX

2.4 Betrieb und Security

2.5 Entwicklergemeinschaft

— Herkunft, Apache Foundation, Entwicklungsphilosophien, Anzahl Entwickler, ... —

2.6 Verwandte Produkte

— Ergänzende oder konkurrierende Produkte —

2.6.1 YARN

2.6.2 Mesos

2.6.3 Flink

3 Spark in der Praxis

Im Folgenden wird Apache Spark im Rahmen zweier grundsätzlich verschiedener Anwendungsfälle betrachtet.

Beispiel 1: Eine typische Anwendung mit verteilten lokalem Storage (HDFS) und Spark als „Client“ eines bestehenden Yarn Clustermanagers. — Commodity Hardware (Rasperry Pi Cluster). —

Beispiel 2: Eine untypische Anwendung mit verteiltem entfernten Storage und dem Spark Standalone Clustermanager. — HPC Hardware („Thunder“ des Hamburger KlimaCampus). —

3.1 Echtzeitbewertung von Twitter-Accounts nach ihrer Relvanz für Spark

— Fusion von Tweets und Mailinglisten https://spark.apache.org/docs/1.3.0/mllib-feature-extraction.html Implementation auf einem Raspberry Pi Cluster mit HDFS und Yarn Clustermanager —

3.1.1 Beschreibung des Problems

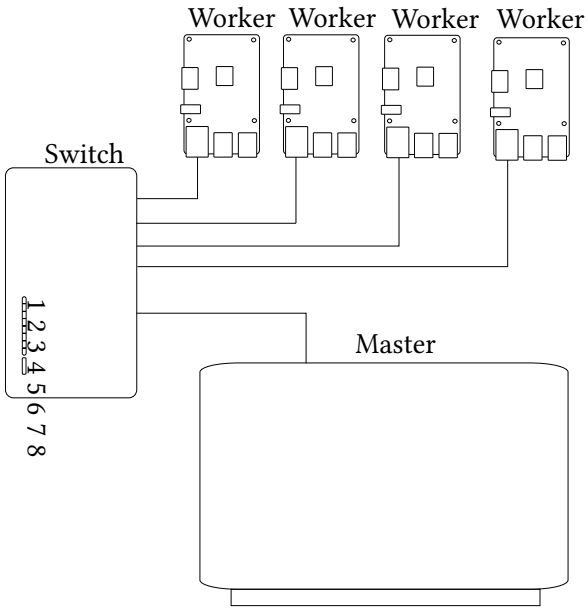
— Es sollen die beiden Spark Mailingslisten (Developer, User) zur Identifikation relevanter und aktueller Themen genutzt werden. Mit den so bewerteten Begriffen können wiederum Tweets bewertet werden. Mit den Tweets können dann ganze Accounts nach ihrer Relevanz beurteilt werden. —
— Zwei Datenquellen: Tweets (Nahe-Echtzeit), Entwickler-Emails (Sporadisch) —
— Stichworte: HDFS, Yarn, Rasperri Pi Cluster, Machine Learning, Feature Extraction, Big Data Life Cycle —

3.1.2 Hardwarekontext und Performance-Basisdaten

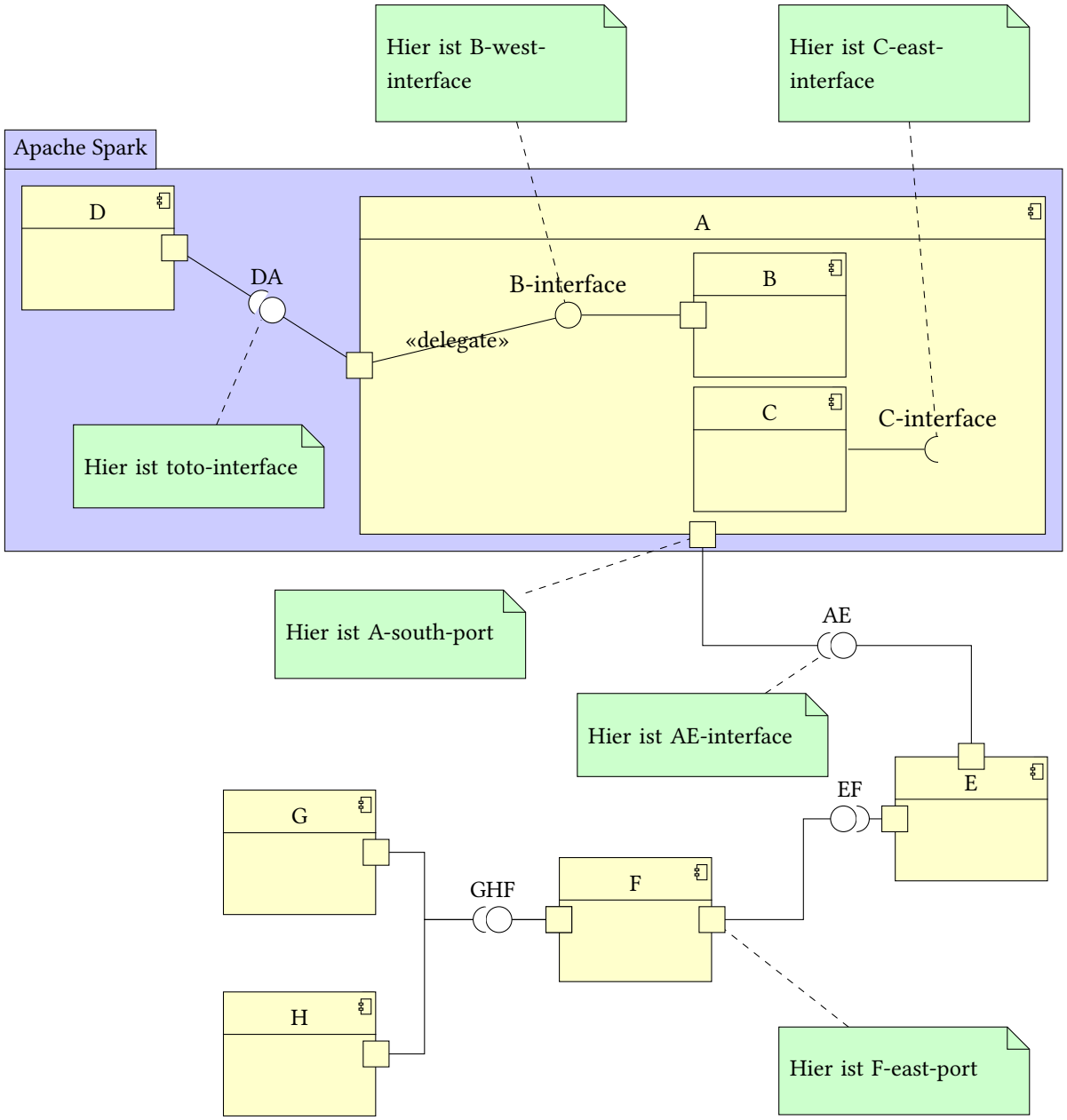
— hier kommen die eingesetzten systeme, und relevante laufzeitmessungen (netzwerk, storage, cpu) hin —

Tabelle 3.1: —DUMMY— Netzwerkdurchsatz

Nachrichtengröße	Worker $\rightarrow Worker$	Master $\rightarrow Worker$	Worker $\rightarrow Master$
1kB	50ms	837ms	970ms
64kB	47ms	877ms	230ms
1MB	31ms	25ms	415ms
64MB	35ms	144ms	2356ms


Worker
Worker
Worker
Worker
Switch
1
2
3
4
5
6
7
8
Master

3.1.3 Architekturübersicht

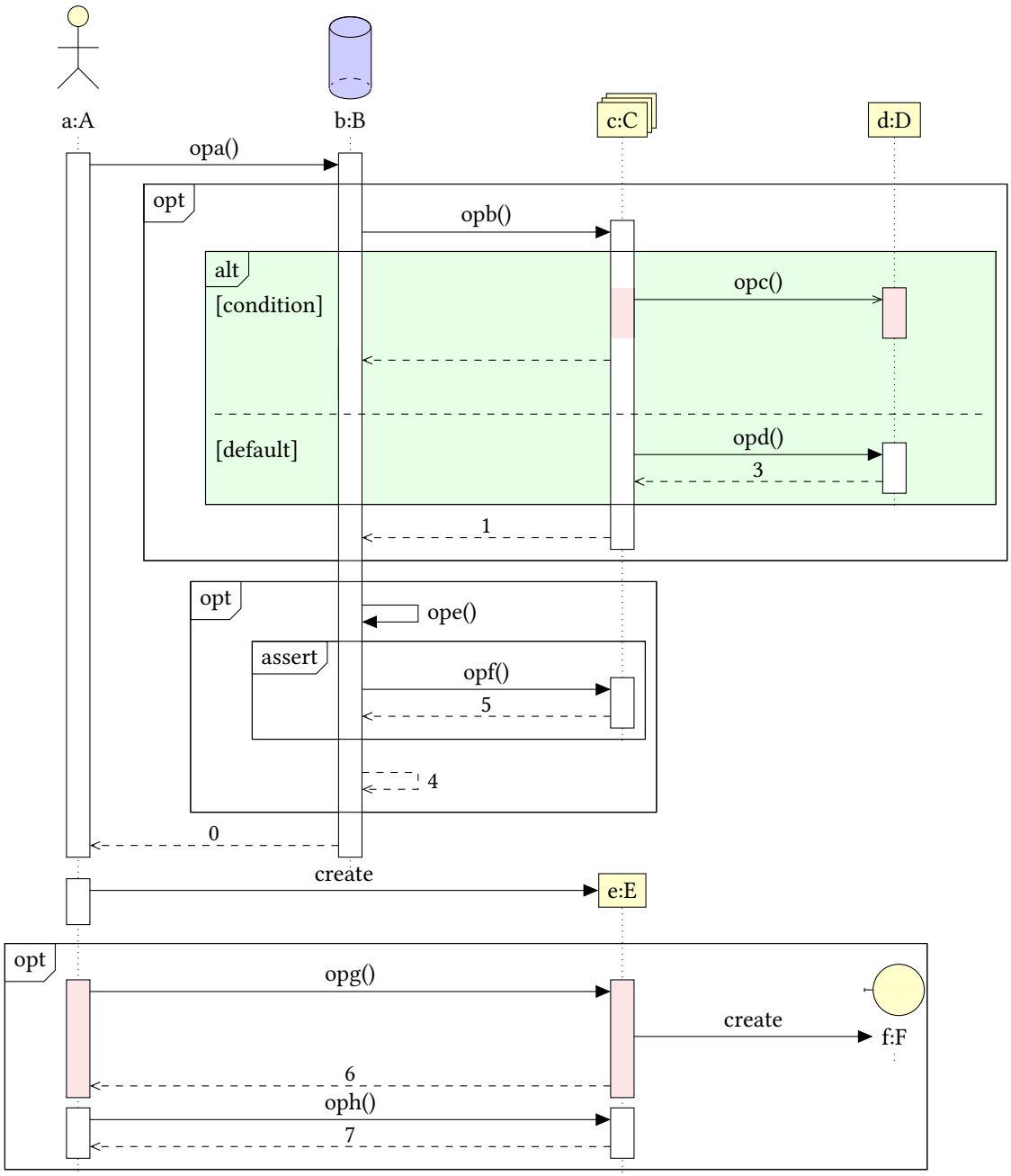


3.1.4 Detailierte Lösungsbeschreibung

— hier kommen diagramme und codeschnipsel hin —

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object ScalaApp {
  val my_spark_home = "/home/daniel/projects/spark-1.1.0"

  def main(args: Array[String]): Unit = {
    val logFile = my_spark_home + "/README.md"

    val conf = new SparkConf().setAppName("ScalaApp")
    val sc = new SparkContext(conf)

    val parList1 = sc.parallelize(List(1,2,3,4,5,6))
    val parList2 = sc.parallelize(List(5,6,7,8,9,10))
    val str1 =
      "RDD1: \%s".format(parList1.collect().deep.mkString(" "))
    val str2 =
      "RDD2: \%s".format(parList2.collect().deep.mkString(" "))
    val str3 =
      "# of RDD1: \%s".format(parList1.count())
    val str4 =
      "Intersect: \%s".format(parList1.intersection(parList2).collect()
    val str5 =
      "Intersect: \%s".format(parList1.cartesian(parList2).collect()
  }
}
```

Listing 3.1: Treiber für Testanwendung (Programmiersprache Scala)



3.1.5 Ergebnisse

— Tabellen und Diagramme Ergebnissen, evt. Skalierungsverhalten — — Bewertung —

3.2 Evaluierung einer spark-basierten Implementation von CDOs auf einem HPC Cluster mit nicht-lokalem Storage

— Implementation ausgewählter CDOs (sehr wenige, möglicherweise nur 1-2) mit der Core-API von Spark. Testlauf auf einem HPC Cluster mit nicht-lokalem, allerdings per Infiniband angeschlossenen Storage. Insbesondere Betrachtung des Skalierungsverhaltens und der „Sinnhaftigkeit“. —

3.2.1 Beschreibung des Problems

— Erläuterung von CDOs (Climate Data Operators). —

3.2.2 Hardwarekontext und Performance-Basisdaten

— hier kommen die eingesetzten systeme, und relevante laufzeitmessungen (netzwerk, storage, cpu) hin —

3.2.3 Architekturübersicht

— hier kommen Verteilungs- und Komponentendiagramm hin —

3.2.4 Detailierte Lösungsbeschreibung

— hier kommen laufzeitdiagramme und codeschnipsel hin —

3.2.5 Ergebnisse

— Tabellen und Diagramme Ergebnissen, evt. Skalierungsverhalten — — Bewertung —

4 Schlussbetrachtung

4.1 Kritische Würdigung der Ergebnisse

4.2 Ausblick und offene Punkte

See also [Woo01].

Literatur

[DG04] Jeffrey Dean und Sanjay Ghemawat. “MapReduce: Simplied Data Processing on Large Clusters”. In: *OSDI* (2004).

[Goo] Google. *Google Trends*. URL: https://www.google.com/trends.

[Lan01] Doug Laney. “3D Data Management: Controlling Data Volume, Velocity and Variety”. In: *Application Delivery Strategies* (2001).

[MH99] Karl Knight Max Hailperin Barbara Kaiser. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. 1999, 278ff.

[Pag01] Lawrence Page. *Method for node ranking in a linked database*. US Patent 6,285,999. 2001. URL: http://www.google.com/patents/US6285999.

[SG03] Shun-Tak Leung Sanjay Ghemawat Howard Gobioff. *The Google File System*. Techn. Ber. Google, 2003.

[SR14] Dilpreet Singh und Chandan K Reddy. “A survey on platforms for big data analytics”. In: (2014).

[Woo01] W. K. Wootters. “Entanglement of formation and concurrence”. In: *Quantum Information and Computation* 1 (2001), S. 27–47.

sample

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Januar 2345 Daniel Kirchner