

Kurzfassung

Die Verarbeitung und Analyse von Klimadaten umfassen heutzutage größere Datenmengen, die sehr oft strukturiert innerhalb der *NetCDF*-Dateien aufbewahrt werden. Die Verarbeitungsprozesse der Datenanalyse benötigen komplexe leistungsfähige Systemen mit größerem Berechnungspotential, um die Datenverarbeitung in akzeptabler Zeit ausführen zu können. Moderne Big-Data-Werkzeuge bieten gut strukturierte Plattformen für die Verarbeitung wissenschaftlicher Daten innerhalb der *NetCDF*-Dateien.

In dieser Arbeit werden mögliche Alternativen der Verwendung von Big-Data-Werkzeugen erläutert, die eine Möglichkeit schaffen, die vom Nutzer angeforderte Verarbeitungsabläufe innerhalb einer Weboberfläche auszuführen und die Ergebnisse mit Hilfe einer grafischen Datendarstellung begutachten zu können. Auf der Basis des entwickelten Systems wird untersucht, inwiefern die aktuellen Werkzeuge für interaktive Analyse der Klimadaten geeignet sind. Dabei werden sämtliche Berechnungsprozesse mittels *SciSparks* auf einem Cluster von Berechnungsknoten ausgeführt. Die Steuerung dieser Prozessen sowie Visualisierung der Verarbeitungsergebnisse ermöglicht *Apache Zeppelin* innerhalb einer Webschnittstelle. Es wird untersucht, inwiefern genannte Werkzeuge angeforderte Voraussetzungen bereits erfüllen können. Diese Systeme werden durch einige Komponenten erweitert, um einen Prototyp des vorgestellten Ansatzes zu entwickeln. Somit werden auf der Basis theoretischer Grundlagen die aufgesetzten Komponenten in einem System mit einer Benutzerwebschnittstelle zusammengefasst. Dabei wurde vorhandene *SciSpark*-Funktionalität mit den implementierten *CDO*-Operatoren und dem *Stencil*-Verfahren für ein-, zwei- und dreidimensionale *NetCDF*-Variablen erweitert.

Zum Schluss wird gezeigt, wie effizient eine Ausführung der unterschiedlichen Prozessabläufe in dem entwickelten System sein kann und welche Einschränkungen auf die Software und Hardware ungeeignet beziehungsweise nicht leistungsfähig genug sind.

Danksagung

Mein besonderer Dank geht an meinen Betreuer Dr. Julian Kunkel für die Bereitstellung des Themas, sowie seine hilfreichen Anregungen, wertvollen Ratschläge und die konstruktive Kritik. Ich danke meinen Eltern und meinen Freunden für die allezeit gebotene Unterstützung. Außerdem möchte ich allen anderen danken, die mir während des Studiums und der Abschlussarbeit geholfen haben.

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung..... | 5 |
| 1.1 | Motivation..... | 5 |
| 1.2 | Zielsetzung..... | 6 |
| 1.3 | Aufbau der Arbeit..... | 7 |
| 2 | Hintergrund und verwandte Arbeiten..... | 8 |
| 2.1 | Wissenschaftliche Dateiformate..... | 8 |
| 2.1.1 | HDF..... | 8 |
| 2.1.2 | CDF..... | 9 |
| 2.1.3 | NetCDF..... | 10 |
| 2.2 | Verarbeitung von großen Datenmengen..... | 12 |
| 2.2.1 | Apache Hadoop..... | 12 |
| 2.2.1.1 | Hadoop Distributed File System..... | 13 |
| 2.2.1.2 | Hadoop MapReduce..... | 14 |
| 2.2.1.3 | Hadoop Ökosystem..... | 16 |
| 2.2.2 | Apache Spark..... | 16 |
| 2.2.3 | SciSpark..... | 21 |
| 2.2.3.1 | SciTensor..... | 23 |
| 2.2.3.2 | SciDataset..... | 24 |
| 2.2.3.3 | Zusammenfassung..... | 25 |
| 2.2.4 | Apache Zeppelin..... | 25 |
| 2.2.5 | Jupyter..... | 27 |
| 2.2.6 | OPeNDAP..... | 28 |
| 2.2.7 | CDO: Climate Data Operators..... | 29 |
| 2.2.8 | Stencil Algorithmen..... | 31 |
| 3 | Design..... | 33 |
| 3.1 | Methodik..... | 33 |
| 3.1.1 | Auswahl des Cluster-Computing-Frameworks..... | 34 |
| 3.1.2 | Auswahl der interaktiven Benutzeroberfläche..... | 36 |
| 3.1.3 | Datenquellen..... | 37 |
| 3.1.4 | Grundlegende Architektur..... | 37 |
| 3.1.5 | Auswahl der Programmiersprachen und Bibliotheken..... | 37 |
| 3.1.5.1 | Programmiersprache für die Cluster-Computing-Komponente..... | 38 |
| 3.1.5.2 | Programmiersprache für die Datenvisualisierung..... | 39 |

| | |
|---|-----------|
| 3.1.5.3 Auswahl der JavaScript-Bibliothek..... | 39 |
| 3.2 Datenstrukturen und Datenverarbeitung..... | 41 |
| 3.2.1 CdoDataset Datenstruktur..... | 41 |
| 3.2.2 Variable Datenstruktur..... | 42 |
| 3.2.3 JSON-Struktur der Datenvisualisierung..... | 43 |
| 3.2.4 Datenverarbeitung mit CDOs..... | 43 |
| 3.2.5 Datenverarbeitung mit Stencil..... | 47 |
| 3.3 Ablauf..... | 47 |
| 3.3.1 Datenanfrage an eine lokale Datenquelle..... | 47 |
| 3.3.2 Datenanfrage an eine Remotedatenquelle..... | 48 |
| 3.3.3 Ablauf der Datenverarbeitung..... | 48 |
| 4 Implementation..... | 51 |
| 4.1 CDO-Merkmale..... | 51 |
| 4.2 Stencil Algorithmen..... | 53 |
| 4.3 Datenvisualisierung..... | 54 |
| 4.4 Parallele Datenverarbeitung..... | 56 |
| 5 Evaluation..... | 57 |
| 5.1 Testumgebung..... | 57 |
| 5.2 Vorgehensmodell und Experimente..... | 57 |
| 5.2.1 Experiment 1: Verarbeitung der CDO-Szenarios..... | 60 |
| 5.2.2 Experiment 2: Stencil Algorithmus..... | 61 |
| 5.2.3 Experiment 3: Datenvisualisierung..... | 61 |
| 5.3 Messergebnisse..... | 61 |
| 5.3.1 Experiment 1: Verarbeitung der CDO-Szenarios..... | 61 |
| 5.3.2 Experiment 2: Stencil Algorithmus..... | 62 |
| 5.3.3 Experiment 3: Datenvisualisierung..... | 63 |
| 5.3.4 Analyse der Verarbeitungszeiten..... | 63 |
| 5.4 Fazit..... | 64 |
| 6 Zusammenfassung und Ausblick..... | 66 |
| 6.1 Zusammenfassung..... | 66 |
| 6.2 Ausblick..... | 66 |
| Abbildungsverzeichnis..... | 70 |
| Listingübersicht..... | 71 |
| Tabellenverzeichnis..... | 72 |

1 Einleitung

In diesem Kapitel wird ein erster Einblick in diese Arbeit gegeben. In Abschnitt 1.1 wird zunächst auf die thematische Einführung eingegangen. Darauf werden in Abschnitt 1.2 die Ziele der Arbeit dargestellt. Zum Schluss erfolgt in Abschnitt 1.3 die kurze Beschreibung des Aufbaus der Arbeit.

1.1 Motivation

In den letzten Jahren ist das wissenschaftliche Datenaufkommen erheblich gewachsen und steigt weiterhin rasant. Deswegen ist die Entwicklung von leistungsstarken Systemen für die Analyse großer Datenmengen in dazu passenden Datenstrukturen sehr wichtig geworden.

Es wurden inzwischen mehrere von Plattformen und Applikationen unabhängige Dateiformate entwickelt, die effizient und strukturiert mehrdimensionale Daten größerer Mengen speichern können. Zu der Liste gehören CDF, NetCDF und HDF Dateiformate. Die gesammelten Informationen werden oft in Cloud-Infrastrukturen abgelegt, um sie beispielsweise weltweit verfügbar zu machen.

Für eine Verarbeitung der Daten in der wissenschaftlichen Welt benutzt man sowohl grafische Anwendungen wie integrierte *Matlab*- und *Octave*-Entwicklungsumgebungen als auch Konsolenanwendungen wie zum Beispiel die *Climate Data Operators (CDO)*.¹ Gleichzeitig spielen in der Analyse größerer Dateien spezifische Bibliotheken in verschiedenen Programmiersprachen eine große Rolle, die auf statistische Aufgaben ausgelegt sind. Zu der Gruppe solcher Programmiersprachen gehören *R*, *Python*, *Matlab*, *IDL (Interactive Data Language)*.

Heutzutage wird in der Klimaforschung sehr häufig das *NetCDF Dateiformat* verwendet, um die Messergebnisse strukturiert zu speichern, und mit der Hilfe von der Konsolenanwendung *CDO* zu analysieren und zu bearbeiten.

Gleichzeitig werden von großen Softwareunternehmen Big-Data-Werkzeuge wie Hadoop, Hive und Spark entwickelt, die eine Datenverarbeitung größerer Datensätze auf einem Cluster von mehreren Computern mit einfachen Programmiermodellen ermöglichen. Die Struktur solcher Werkzeuge ermöglicht die Berechnungen und Speicherungen von einem

¹ Die *Climate Data Operators (CDO)* ist eine Kollektion von mehreren Operatoren, die für eine Standardverarbeitung von Klima- und Prognosemodelldaten verwendet werden [CDO16].

einzelnen Server bis zu Tausenden von Maschinen zu skalieren. Heutzutage sind in der Geschäftswelt diese Systeme sehr populär.

Im Kontext dieser Arbeit wird untersucht in wieweit eines dieser Big-Data-Frameworks für die Analyse von wissenschaftlichen Daten in dem NetCDF Dateiformat adaptiert werden kann. Im zweiten Kapitel „Hintergrund und verwandte Arbeiten“ werden sowohl Vorteile als auch Nachteile dieser Systeme detailliert erläutert.

Der Grundgedanke dieser Arbeit besteht darin, die Berechnungsprozesse der Analyse von den *NetCDF* Datensätzen mit Hilfe des *Cluster-Computing-Systems Apache Spark* vorzunehmen.

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht in der Untersuchung der traditionellen Analyseprozesse von Klimadaten und darin, wie diese Prozessen mit Hilfe von Big-Data-Frameworks effizient umgesetzt werden können. Hierfür wird das *SciSpark*-Framework erweitert um die Ausführung von Analyseprozessen auf die Klimadaten zu ermöglichen. Im Folgenden lassen sich allgemeine Anforderungen für das Analysesystem formulieren:

- Interaktiver Zugriff (d.h. Benutzer haben eine einfache und klar definierte Bildschirmschnittstelle für Befehleingaben und für die Systemsteuerung), welcher auf Eingaben agiert.
- Ausführungsbefehle vergleichbar den traditionellen Analyseprozessen (d.h. bestehende Workflows können direkt oder mit wenig Anpassung im neuen Werkzeug genutzt werden).
- Daten im *NetCDF4*-Format sollen verarbeitet werden.

Die initiale Festlegung der Zielplattform ergab:

- CDO als Benutzerwerkzeug zur Formulierung der Eingaben.
- *Stencil Algorithmen*¹ mit einer benutzerdefinierten Berechnungsfunktion von räumlichen Elementen (d.h. Nutzer können eine Funktion definieren, die für die Aktualisierung von Datenelementen nach einem bestimmten Muster angewendet wird).
- Dreidimensionale Darstellung der Datenverarbeitungen

¹ *Stencil Algorithm* – ist eine Klasse iterativer Kerneln (Funktionen), die Datenelemente nach einem festen Muster, genannt Schablone, aktualisieren.

1.3 Aufbau der Arbeit

Im zweiten Kapitel werden für diese Arbeit wesentliche theoretische Grundlagen erläutert und verwandte Projekte vorgestellt. Hierbei werden sowohl Dateiformate, welche die Datenstruktur von Messergebnissen speichern, als auch Systeme für deren Bearbeitung und Analysen vorgestellt. Im dritten Kapitel werden der strukturelle Aufbau des entwickelten Systems und zu verwendende Komponenten für die Datenstruktur und für die Funktionalität der Anwendung beschrieben. Das vierte Kapitel beschreibt die Implementierung von angewendeten Lösungen und Techniken. Anschließend wird im fünften Kapitel eine Untersuchung der Leistungsstärke des Systems präsentiert und die Eignung der Lösung für die iterative Datenanalyse untersucht. Zum Schluss fasst der sechste Teil die Untersuchung mit einem Ausblick auf die Entwicklungsmöglichkeiten zusammen.

2 Hintergrund und verwandte Arbeiten

In diesem Kapitel werden wichtige theoretische Grundlagen wie Dateiformate und Frameworks verteilter Verarbeitung von großen Datensätze erläutert. Zunächst werden in Abschnitt 2.1 einige Dateiformate vorgestellt, die in der Wissenschaft verwendet werden. Hierbei werden HDF, CDF und NetCDF detailliert erläutert. Danach werden in Abschnitt 2.2 Frameworks und Komponenten beschrieben, die für die Ausführung der Analyseprozesse von wissenschaftlichen Daten dienen. Dabei werden die Big-Data-Frameworks Apache Hadoop, Apache Spark und SciSpark sowie die interaktiven Notebook-Webanwendungen Apache Zeppelin und Jupyter vorgestellt. Zum Schluss des Abschnittes werden die verteilte Datenarchitektur OPeNDAP und das traditionelle Analysewerkzeug der Klimadaten CDO präsentiert.

2.1 Wissenschaftliche Dateiformate

Für die Speicherung der Messergebnisse wurden in der wissenschaftlichen Welt einige Dateiformate entwickelt, die große Datenmengen unterstützen. In diesem Abschnitt werden in der Klimaforschung verwendbare Dateiformate und entsprechende Softwarebibliotheken erläutert, welche praktische Schnittstellen für die Datenmanipulation zur Verfügung stellen.

2.1.1 HDF

Das *Hierarchical Data Format (HDF)* besteht aus einem Datenmodell, einer Bibliothek und eines Dateiformats für das Speichern und Verwalten von Daten. *HDF* wurde ursprünglich von National Center for Supercomputing Applications in Urbana, Illinois/USA, entwickelt. Heutzutage wird es von der *HDF*-Group unterstützt und weiterentwickelt. Das Dateiformat unterstützt eine Vielfalt an Datentypen und ist flexible für die effiziente Ein- und Ausgabe sowie für umfangreiche und komplexe Daten ausgelegt. *HDF* ist ein selbstbeschreibendes Dateiformat und enthält entsprechend neben den Nutzdaten auch Metadaten. Die aktuelle Version des Datenformats und des Datenmodells ist *HDF5*.

Eine *HDF*-Datei besteht hauptsächlich aus folgenden Objekten:

- *Groups (Gruppen)*: Eine Gruppe schließt unterschiedliche *HDF*-Objekte ein, welche mit den Metadaten genauer beschrieben werden.
- *Datasets (Datensätze)*: Datensätze umfassen multidimensionale Arrays der Nutzdaten der Datei. Die Information über die Datensätze wird auch in Metadaten gespeichert.

HDF wird von vielen kommerziellen und nicht kommerziellen Softwareplattformen wie Java, MATLAB¹, Octave², IDL³, Python, R und anderen unterstützt. Die frei verfügbare *HDF*-Werkzeuge bestehen aus Bibliotheken, Befehlszeilenprogrammen, Testquellen, Java-Schnittstellen und einem javabasierten *HDF*-Viewer (*HDFView*). Dieses Format wird für die Speicherung großer Datenmengen in der Geowissenschaft verwendet. [*HDFGr*]

2.1.2 CDF

Das *Common Data Format (CDF)* ist eine Datenabstraktion für eine Speicherung, eine Manipulierung und ein Zugriff auf mehrdimensionale Datensätze. Die Entwicklung von *CDF* wurde vom National Space Science Data Center (NSSDC) der NASA in 1985 gestartet und wird weitergeführt.

Die grundlegende Komponente von *CDF* ist eine Software-Programmierschnittstelle, die eine geräteunabhängige Ansicht des *CDF*-Datenmodells ermöglicht. Durch eine Isolierung seitens der Anwendungsentwickler vom physischen Dateiformat sind konzeptuelle Simplität, eine Geräteunabhängigkeit und zukünftige Erweiterbarkeit möglich geworden.

CDF stellt einen transparenten Zugriff auf Daten und Metadaten durch eine Software-Programmierschnittstelle zur Verfügung. Die *CDF*-Datenstruktur integriert eine Unterstützung von Datenkomprimierung in den Formaten *gZip*⁴, *Run-Length Encoding*⁵ (*RLE*), *Huffman Kodierung*⁶ und die Datenunkomprimierung mit einer Checksumme. Eine sehr wichtige Charakteristik dieses Dateiformats ist die Unterstützung großer Dateien (über 2 Gbyte).

Die aktuell letzte Version dieser Datenabstraktion ist *CDF V3.6.3*. In dieser Version wurden von der NASA Lese- und Schreibschnittstellen für C, FORTRAN, Java, Perl, C#, Visual

1 <https://de.mathworks.com/products/matlab.html>.

2 <https://www.gnu.org/software/octave/>.

3 <http://www.harrisgeospatial.com/ProductsandSolutions/GeospatialProducts/IDL.aspx>.

4 <http://www.gzip.org/>.

5 https://en.wikipedia.org/wiki/Run-length_encoding.

6 https://en.wikipedia.org/wiki/Huffman_coding.

Basic, IDL, MATLAB bereitgestellt. Von Dritt-Entwicklern wurden auch Schnittstellen für Python, Sybase und mySQL angeboten. [CDFnasa]

2.1.3 NetCDF

Das *Network Common Data Format (NetCDF)* ist ein Satz von Softwarebibliotheken und selbstbeschreibenden, maschinenunabhängigen Dateiformaten, die die Erstellung, den Zugriff und die gemeinsame Nutzung von arrayorientierten wissenschaftlichen Daten unterstützen. *NetCDF* wurde von der *Unidata Community (Unidata)* entwickelt und wird von der *University Corporation for Atmospheric Research (UCAR)* betreut. [UCAR]

Von der *UCAR* wurden Softwarebibliotheken, die Lese- und Schreibzugriffe für *NetCDF* Dateien bieten, in *C* und *Java* Programmiersprachen implementiert. Jedoch wurden zu der *C*-Bibliothek mehrere Schnittstellen in *C++*, *Fortran*, *Python*, *R*, *Matlab*, *Perl*, *Ruby*, *IDL* und *Octave* geschrieben, sodass die Bibliothek von allen aktuellen Plattformen angewendet werden kann.

Das *NetCDF*-Dateiformat basiert auf dem *Common Data Format (CDF)* der NASA. Das *NetCDF* Format wurde dementsprechend weiterentwickelt und ist nicht mehr mit dem *CDF* kompatibel. In Kapitel 2.2.2 „*CDF*“ wird dieses detaillierter erläutert.

Die Selbstbeschreibung des *NetCDF* Dateiformats bedeutet, dass in jeder Datei durch die Metadaten eine eigene Struktur der Speicherung von Nutzdaten definiert wird.

Im Laufe dieser Arbeit werden von *Unidata Community* folgende *NetCDF*-Formate vorgestellt [UCAR]:

- das *classic Format*
- das *64-Bit Offset Format*
- das *NetCDF-4 Format*
- das *NetCDF-4 classic Model Format*

Das *classic Format* wurde 1989 entworfen und als einziges *NetCDF*-Dateiformat von *Unidata Software* bis zum Jahr 2004 angewendet. Zudem wird das klassische Format bis zum heutigen Tag als Standardformat in *Unidata*-Anwendungen für die Erstellung neuer Dateien verwendet.

2004 wurde das *64-Bit-Offset-Format* entwickelt. Das *NetCDF-64-Bit-Offset-Format* unterscheidet sich von dem klassischen Format nur in der *VERSION-BYTE*-Sektion und in einem der internen Offsets. Die *64-Bit-Version* hat in der *VERSION-BYTE*-Sektion statt klassischem '\x01' Wert einen '\x02' und das Offset vom Dateianfang ist 64 Bit im Vergleich zu 32 Bits des *32-Bit-Offset-Formats*. Diese kleine Formatänderung erlaubt viel größere Dateien bis zu 8 EiB. Es gibt aber noch einige praktische Größenbeschränkungen und zwar

jede Variable mit einer festen Größe und ein Datensatz innerhalb einer Variablen sind nach wie vor auf 4 GiB begrenzt. Die Begründung für diese Beschränkung besteht darin, einen zusammenfassenden Zugriff auf alle Daten der *NetCDF-Variablen* und alle Datensätze der *NetCDF-Datei* auf 32-Bit-Plattformen zu ermöglichen.

Das *NetCDF-4* Format unterstützt einer Datenkompression, mehrere unbegrenzte Dimensionen, komplexere Datentypen und erreicht eine bessere Leistung. Dies wird durch eine Schichtung der erweiterten *NetCDF-Zugriffsschnittstellen* auf Basis des *HDF5-Formates* erreicht. Das erwähnte *HDF5-Format* wird in Abschnitt 2.1.2 näher erläutert. [UCAR]

Das *NetCDF-4 classic Model Format* wurde gleichzeitig mit *NetCDF-4* bereit gestellt. Dieses Format wurde entwickelt, um die Leistungsvorteile vom *HDF5-Dateiformat* ohne die Komplexität der neuen Programmierschnittstelle oder das verbesserte Datenmodell zu verwenden. [UCAR]

```
1 netcdf /path/to/dataset/atls14-CyG11B.nc {
2   dimensions:
3     longitude = 480;
4     latitude = 241;
5     time = UNLIMITED; // (1096 currently)
6   variables:
7     float longitude(longitude=480);
8       :units = "degrees_east";
9       :long_name = "longitude";
10
11     float latitude(latitude=241);
12       :units = "degrees_north";
13       :long_name = "latitude";
14
15     int time(time=1096);
16       :units = "hours since 1900-01-01 00:00:0.0";
17       :long_name = "time";
18       :calendar = "gregorian";
19
20     short sf(time=1096, latitude=241, longitude=480);
21       :scale_factor = 7.376412457340497E-7; // double
22       :add_offset = 0.024169553051021742; // double
23       :_FillValue = -32767S; // short
24       :missing_value = -32767S; // short
25       :units = "m of water equivalent";
26       :long_name = "Snowfall";
27       :standard_name = "lwe_thickness_of_snowfall_amount";
...

```

Auflistung 2.1. Ausgabe des *ncdump*¹-Kommando für eine NetCDF Datei.

¹ *ncdump* – ein *CDO*-Kommando, welches eine Textrepräsentation der Datenstruktur einer *NetCDF-Datei* erstellt.

In Auflistung 2.1 wird einen Ausschnitt einer Beispielstruktur einer NetCDF Datei dargestellt. Dieser Ausschnitt zeigt eine Dateistruktur durch eine Beschreibung von Dimensionen und Variablen. Die Struktur der NetCDF-Datei definiert drei vorhandene Dimensionen *longitude*, *latitude*, *time*. Die Variablen *longitude*, *latitude*, *time* und *sf* werden durch eigene Attribute beschrieben. Am Beispiel der *sf*-Variable (Zeile 20) kann man sehen, dass die Werte dieser Variable einen *short*-Datentyp haben und dreidimensional sind. Durch das *standard_name*-Attribut sieht man die Bedeutung des Wertes. Das Attribut *units* beschreibt die Maßeinheiten des Wertes, welche in diesem Fall eine entsprechende Wasserhöhe der Niederschlagsmessungen in Millimeter zeigen.

2.2 Verarbeitung von großen Datenmengen

In diesem Abschnitt werden Frameworks und Komponenten beschrieben, die für effiziente Analysen von wissenschaftlichen Daten großer Mengen dienen. Dabei werden weit verbreitete Cluster-Computing-Systeme und interaktive Umgebungen sowie Werkzeuge für traditionelle Analyseprozesse der Klimadaten vorgestellt.

2.2.1 Apache Hadoop

Apache Hadoop ist eine verteilte Dateninfrastruktur für zuverlässige und skalierbare Verarbeitung und Analyse großer Datenmengen über mehrere Knoten innerhalb eines Clusters von hauptsächlich *Commodity*¹-Servern. *Apache Hadoop* beinhaltet auch Werkzeuge für die parallele und verteilte Verarbeitung größerer Datensätze auf dem Cluster von Servern mit *MapReduce*-Programmiermodell. Alle Module des *Hadoop*-Systems sind mit der Annahme aufgebaut worden, dass Hardware-Ausfälle ein häufiges Ereignis ist, welches automatisch vom System behandelt werden muss. *Apache Hadoop* teilt Dateien in große Blöcke und verteilt diese Blöcke über die Knoten in einem Cluster.

Das System überträgt auch verpackte Programmcodes, die für die Datenverarbeitung benötigt werden, in die Knoten des Clusters, um verteilte Daten parallel zu verarbeiten.

Die Datenverarbeitung nutzt den Vorteil der Datenlokalität: die Knoten manipulieren nur die Datensätze, die sie zur Verfügung haben. Dieser Vorteil schafft den Datensätzen eine Möglichkeit, schneller und effizienter als auf einer konventionellen Cluster-Architektur aus *COTS*²-Komponenten bearbeitet zu werden. [ASFHadoop]

¹ *Commodity*-Servern sind günstige Computing-Komponenten, die im gegebenen Infrastem vorhanden sind und für paralleles Computing verwendet werden [https://en.wikipedia.org/wiki/Commodity_computing].

² *Commercial off-the-shelf* oder auch *Components-off-the-shelf* (englisch für Kommerzielle Produkte aus dem Regal), kurz *COTS*, werden seriengefertigte Produkte aus dem Elektronik- oder Softwaresektor bezeichnet, die in großer Stückzahl völlig gleichartig aufgebaut und verkauft werden. https://de.wikipedia.org/wiki/Commercial_off-the-shelf

Apache Hadoop besteht aus vier folgenden Module:

- *Hadoop Common* sind die gemeinsamen Dienstprogramme für die Unterstützung anderer Module
- *Hadoop Distributed File System (HDFS)* ist ein verteiltes Dateisystem, welches einen leistungsfähigen Zugriff auf die Anwendungsdaten zur Verfügung stellt
- *Hadoop YARN* ist ein Framework für die Jobplanung und das Cluster-Ressourcen-Management.
- *Hadoop MapReduce* ist ein *YARN*-basiertes System für die parallele Verarbeitung großer Datensätze basierend auf dem *Map-Reduce*-Paradigma.

Das *Apache Hadoop*-Framework wurde zum größten Teil in der Programmiersprache *Java* geschrieben, wobei einige native Codes in der Programmiersprache *C* und Befehlszeilenprogramme als Shell-Skripte geschrieben worden sind.

2.2.1.1 Hadoop Distributed File System

Besondere Aufmerksamkeit verdient das *Hadoop Distributed File System (HDFS)*. Die *HDFS*-Komponente ist ein hochgradiges fehlertolerantes verteiltes Filesystem, welches für den Einsatz auf einer kostengünstigen Hardware entwickelt wurde. Dieses System stellt einen zuverlässigen Zugang mit einem guten Datendurchsatz zu den Anwendungsdaten mit größeren Datensätzen zur Verfügung. *HDFS* stellt folgende praktische Funktionen bereit:

- *Dateiberechtigung* und *Authentifizierung*. Das *HDFS*-Dateiberechtigungssystem wird gleichartig zu Dateiberechtigungen der Linux-ähnlichen Plattformen gestaltet.
- *Rack-Awareness*. Typische *Hadoop*-Cluster werden mit Hilfe von Racks und Netzwerkverkehren zwischen Clusterknoten mit gleichen Racks organisiert. Dadurch verlagert das System die Replikate von Datenblöcken in verschiedenen Racks um die Fehlertoleranz zu steigern.
- Werkzeuge für eine Systemsdiagnostizierung, eine Delegation von Berechtigungen und eine Datenbalancierung auf dem Cluster.
- *Upgrade* und *Rollback*. *HDFS* erlaubt es nach der Softwareaktualisierung im Fall des Fehlerauftritts das System zu dem vorherigen Stand zurückzurollen.
- *Checkpoint-* und *Backupknoten*. Das *HDFS*-System beinhaltet Werkzeuge für die Überprüfung und Behebung eines Knotenausfalls.

- *Monitoring*. Außerdem bietet *HDFS* eine Weboberfläche für eine Darstellung des aktuellen Status des Clusters sowie der Systemkonfiguration und elementarer Statistiken vom Cluster.

2.2.1.2 Hadoop MapReduce

Als nächste wesentliche Komponente wird das Programmierparadigma von *MapReduce* erläutert. Dieser Teil des *Hadoop*-Frameworks ist zuständig für eine Verarbeitung großer Datensätze mit einem parallelen und verteilten Algorithmus. Wie am Namen zu erkennen ist, enthält der Algorithmus zwei grundlegende Aufgaben: *Map* und *Reduce*. *Map* nimmt einen Satz von Eingangsdaten und wandelt ihn in andere Datensätze um, wobei einzelne Elemente von Eingangsdatensätzen in *Key-Value*-Tupeln aufgespaltet werden. Der *Reduce*-Teil bekommt am Eingang die *Key-Value*-Tupel und vereint sie in kleineren Sätzen von Tupeln. Die Reihenfolge der *Map*- und *Reduce*-Aufgaben entspricht dem Namen des Paradigma: der *Reduce*-Teil wird immer nach dem *Map*-Teil ausgeführt.

Im Folgenden wird an einem Beispiel erklärt, wie das Programmiermodell praktisch angewendet werden können. Dabei wird berechnet, wie oft einzelne Wörter in einer Textdatei aus dem Datenspeicher getroffen werden (im Weiteren *Wortfrequenz* genannt). Die Auflistung 2.2 stellt *Mapper*-Schnittstellen durch eine Implementierung in *Java*-Klassen für die *Hadoop MapReduce*-Komponente dar. Dabei wird eine Klasse *TokenizerMapper* erstellt, die die *Mapper*-Schnittstelle von *Hadoop MapReduce* beschreibt. In der Zeile 4 ist die Funktion *map* implementiert. Hier werden aus dem Inhalt der Textdatei Textzeilen genommen und mit Hilfe eines *StringTokenizer*-Objektes wird die gesamte Zeile auf einzelne Wörter gespalten (Zeile 4). Jedem Wort wird ein Zähler mit dem Wert eins zugewiesen und in dem *Context*-Objekt aufbewahrt (Zeile 8).

```

1 public static class TokenizerMapper extends
    Mapper<Object, Text, Text, IntWritable> {
2     private final static IntWritable one = new IntWritable(1);
3     private Text word = new Text();
4     public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
5         StringTokenizer itr = new StringTokenizer(value.toString());
6         while (itr.hasMoreTokens()) {
7             word.set(itr.nextToken());
8             context.write(word, one);
9         }
10    }
11 }

```

Auflistung 2.2. Implementierung der Mapper-Schnittstelle für Hadoop MapReduce.

Eine *Reducer*-Schnittstelle des *Hadoop MapReduces* lässt sich durch *SumReducer*-Klasse implementieren, welche in der Auflistung 2.3 vorgestellt wird. Dabei werden in der Funktion *reduce* die Werte der Zähler der gleichen Worte summiert (Zeile 6) und die

endgültigen Ergebnisse werden in das *Context*-Objekt gespeichert (Zeile 9), welches letztendlich in dem Datenspeicher abgelegt wird.

```
1 public static class SumReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {
2     private IntWritable result = new IntWritable();
3     public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
4         int sum = 0;
5         for (IntWritable val : values) {
6             sum += val.get();
7         }
8         result.set(sum);
9         context.write(key, result);
10    }
11 }
```

Auflistung 2.3. Implementierung der *Reducer* Schnittstelle für *Hadoop MapReduce*.

Die Kodeauflistung 2.4 zeigt die Konfigurierung des Berechnungsablaufs aus der *Java-main*-Funktion. Hier lassen sich *Mapper*- und *Reducer*-Klassen in ein Arbeitsablauf-Objekt *Job* in Zeilen 4 und 5 übergeben. In der Zeile 7 wird ein Typ der ausgehenden Ergebnissen definiert. Schließlich werden in Zeilen 9 und 10 der Auflistung 2.4 die Pfade zu den Ein- und Ausgangsdateien eingestellt.

```
1 Configuration conf = new Configuration();
2 Job jobCount = Job.getInstance(conf, "word count");
3 jobCount.setJarByClass(ExampleMapReduce.class);
4 jobCount.setMapperClass(TokenizerMapper.class);
5 jobCount.setCombinerClass(SumReducer.class);
6 jobCount.setReducerClass(SumReducer.class);
7 jobCount.setOutputKeyClass(Text.class);
8 jobCount.setOutputValueClass(IntWritable.class);
9 FileInputFormat.addInputPath(jobCount, new Path("/path/to/input"));
10 FileOutputFormat.setOutputPath(jobCount, new Path("/path/to/output"));
11 jobCount.waitForCompletion(true);
```

Auflistung 2.4. Kalkulation der Wortfrequenz in *Hadoop MapReduce*.

Eine Besonderheit der Ausführung von einem *Map-Reduce*-Berechnungsprozess ist die Notwendigkeit der Speicherung von Kalkulationsergebnissen nach jeder Iteration direkt in ein Dateisystem. Die Abbildung 2.1 zeigt einen Ablauf eines iterativen Algorithmus in *Hadoop MapReduce* mit der Verwendung *Hadoop Distributed File System (HDFS)* als Datenspeicher. Hier werden Zwischenergebnisse nach jeder Iteration in *HDFS* gespeichert.

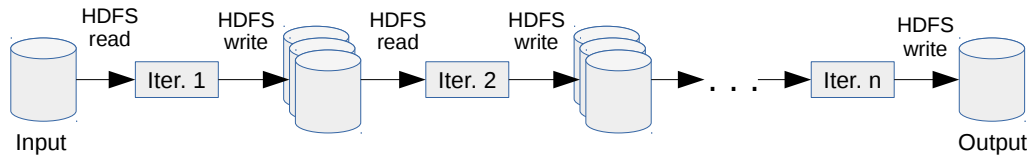


Abbildung 2.1. Ablauf eines iterativen Algorithmus in *Hadoop MapReduce*.

Dadurch kann man sehen, dass die Zwischenergebnisse $n-1$ mal in physischem Speicher abgelegt werden, was entscheidend viel Zeit benötigt.

Sowohl *HDFS* als auch die *MapReduce* Datenverarbeitung sind fehlertolerant. *Hadoop MapReduce* verwendet dafür *TaskTracker*¹-Knoten des Clusters. Diese *TaskTracker*-Knoten stellen *Heartbeats*²-Netzwerkverbindungen zu spezifischen Rechenknoten des Clusters her, an welche vom *JobTracker*³-Dienst erforderliche *MapReduce*-Aufgaben verteilt wurden. Wenn ein *Heartbeat* verloren geht, werden alle in der Bearbeitung stehende Operationen auf andere *TaskTracker*-Knoten verschoben. Dieses Verfahren ist effektiv im Sinne von Fehlertoleranz, hingegen nehmen in einem Fehlerfall die Verarbeitungszeiten zu.

2.2.1.3 *Hadoop Ökosystem*

Schließlich bezieht sich der Begriff *Hadoop* auf ein *Ökosystem* oder eine Sammlung von zusätzlichen Softwarepaketen, die auf oder neben *Apache Hadoop* installiert werden können. Dieses Ökosystem kann Pakete wie *Apache Pig*, *Apache Hive*, *Apache Base*, *Apache Phoenix*, *Apache Spark*, *Apache ZooKeeper*, *Cloudera Impala*, *Apache Flume*, *Apache Sqoop*, *Apache Oozie* und *Apache Storm* beinhalten. [ASFHadoop]

2.2.2 *Apache Spark*

Apache Spark ist ein Open-Source-Cluster-Computing-System für eine über die Knoten des Clusters verteilte Datenverarbeitung. Das Framework entstand als ein Forschungsprojekt. Seit 2013 wird *Spark* von der *Apache Software Foundation* weitergeführt und gewartet. [ASFSpark]

Die grundlegende Idee des *Spark*-Frameworks ist eine besondere Datenstruktur die *Resilient Distributed Dataset (RDD)*. In Abbildung 2.2 wird die *RDD*-Datenstruktur vorgestellt. Hier kann man sehen, dass ein *RDD*-Objekt aus einer Reihe von zu Referenzen untergliederten

¹ *TaskTracker* ist eine Knoten des Clusters in Hadoop Ökosystem, welche die Map-, Reduce- und Shuffle-Aufgaben vom *JobTracker* annimmt [https://wiki.apache.org/hadoop/TaskTracker].

² *Heartbeat* ist eine Netzwerkverbindung zwischen zwei (oder mehr) Rechnern in einem Cluster, um sich gegenseitig darüber zu benachrichtigen, dass sie betriebsbereit sind und ihre Aufgaben noch erfüllen können [https://de.wikipedia.org/wiki/Heartbeat_(Informatik)].

³ *JobTracker* ist ein Dienst innerhalb des Hadoops, der MapReduce-Aufgaben an die spezifischen Knoten des Clusters verteilt [https://wiki.apache.org/hadoop/JobTracker].

Partitionen besteht (Partition 1, Partition 2), welche entsprechende Datensätze aus verteilten Datenquellen (Data Partition 1, Data Partition 2) beinhalten. Jede Partition ist einem Knoten des Clusters zugeordnet.

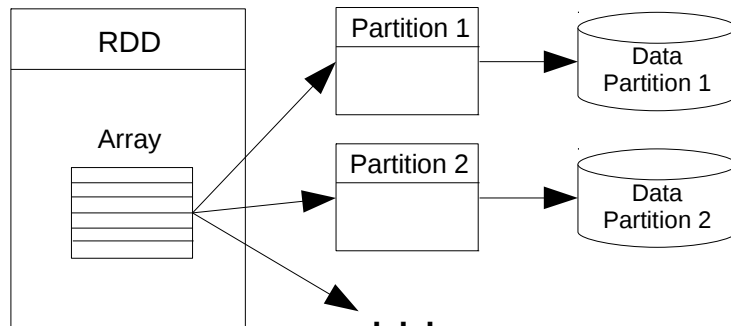


Abbildung 2.2. Apache Spark RDD-Datenstruktur.

Um die Leistungen des Prozessings zu erhöhen, stellt das *Spark*-Framework auf der Basis der *RDD*-Datenstruktur ein *in-Memory-Cluster-Computing* zur Verfügung. Dadurch lässt ein Arbeitsthread eines Knotens Zwischenergebnisse des Verarbeitungsprozesses direkt in den Arbeitsspeicher als Objekte ablegen, was für Arbeitsthreads zugreifbar ist. Somit werden benötigte Zwischenergebnisse viel schneller als in den Systemen mit verteilter Speicher wie *Hadoop MapReduce* erreicht. [AMPLab]

Als ein Vorteil des *Spark*-Systems im Sinne der Leistungsfähigkeit wird eine Unterstützung des *Lazy Evaluation*¹-Verfahrens genannt. Damit spart *Spark* unnötige Berechnungsprozesse und führt nur solche aus, die für gewünschte Ergebnisse relevant sind. Dabei wird zunächst auf eine komplette Liste der Datentransformationen gewartet, und nur dann, wenn die erwünschten Ergebnisse angefragt wurden, werden die entsprechenden Transformationen auf die Daten angewendet.

Das fehlertolerante Verhalten des *Spark*-Systems wird durch die Besonderheiten der *RDD*-Struktur realisiert. Mit diesem Zweck beschreibt jedes *RDD*-Objekt fünf grundlegende Eigenschaften:

1. Eine Liste von Partitionen
2. Eine Verarbeitungsfunktion für jede Datenspalte
3. Eine Liste von Abhängigkeiten zu anderen *RDD*-Objekten
4. Eine Partitionierung für hash-partitionierte *Key-Value-RDDs*
5. Eine Liste von bevorzugten Speicherorten der Verarbeitung von gesplatteten Daten

¹ *Lazy Evaluation* ist eine Art der Auswertung von Ausdrücken, bei der das Ergebnis des auszuwertenden Ausdrucks nur so weit berechnet wird, wie es gerade benötigt wird
[https://de.wikipedia.org/wiki/Lazy_Evaluation]

Mit Hilfe dieser Eigenschaften werden grundlegende Informationen über die Datenherkunft direkt in RDD-Objekten gespeichert. Durch ein Caching von Datensätzen nach Verarbeitungsoperationen im Arbeitsspeicher lassen sich die Ergebnisse innerhalb der *RDD*-Objekte persistent bleiben. Dieses Verfahren ermöglicht es, eingehende Datenverarbeitungen bis zu Faktor zehn im Vergleich zu *Hadoop MapReduce* zu beschleunigen. Das Caching von Spark ist fehlertolerant: wenn irgendeine Partition verloren geht, wird sie automatisch von ursprünglichen RDD-Transformationen erneut berechnet.

Um einen Beispielablauf von *Spark* zu präsentieren, wird in der Auflistung 2.5 ein Codeausschnitt der Kalkulation der Wortfrequenz mit Hilfe vom *Apache Spark*-System vorgestellt. Dieser Ablauf wurde zuvor im Abschnitt 2.2.1 dieser Arbeit für die *Hadoop MapReduce*-Komponente erläutert. Der Codeausschnitt der Auflistung 2.5 wird in der *Scala*¹-Programmiersprache implementiert. Dabei wird dasselbe *Map-Reduce*-Verfahren wie in dem Beispiel des Abschnitts 2.2.1 angewendet. Dabei wird in der Zeile 2 zeilenweise eine Textdatei abgelesen und der Zeileninhalt in einem *RDD[String]*-Array gespeichert, wobei sich in jedem *String*-Objekt eine Zeile der Textdatei speichern lässt. Danach werden in der Zeile 3 die *String*-Objekte des *RDD*s in Wörter gesplittet. Schließlich wird die Kalkulation der Wortfrequenz mit Hilfe der *map*- und *reduceByKey*-Funktionen durchgeführt (Zeilen 4 und 5 der Auflistung 2.5). Die Ergebnisse werden als ein *RDD*-Array von *Key-Value-Paaren* „*RDD[(String, Int)]*“ zurückgegeben.

```

1 val sc = new SparkContext(conf)
2 val textFile = sc.textFile("/path/to/file/textfile.txt")
3 val counts = textFile.flatMap(line => line.split(" "))
4     .map(word => (word, 1))
5     .reduceByKey(_ + _)
6     .cache()

```

Auflistung 2.5. Kalkulation der Wortfrequenz in *Spark*.

Ein Ablaufmodell eines iterativen Algorithmus wird in Abbildung 2.3 präsentiert. Hier lässt sich eine Ausführung des Algorithmus mit *n* Iterationen schematisch darstellen. Am Anfang des Ablaufs werden Datensätze aus dem physischen Speicher, in dem Fall aus *HDFS*, abgelesen. Danach werden Zwischenergebnisse bis zu der letzten Iteration in verteilten Arbeitsspeichern als Objekte aufbewahrt. Schließlich speichert das System am Ende des Arbeitstreads das endgültige Ergebnis in *HDFS*.

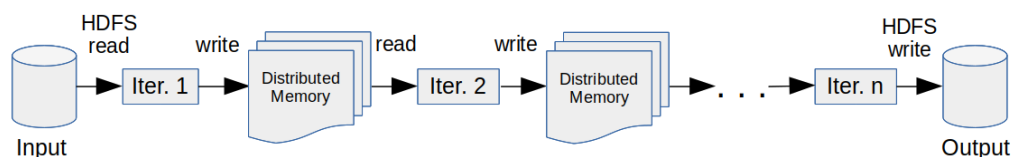


Abbildung 2.3. Ablauf eines iterativen Algorithmus in *Apache Spark*.

¹ <https://www.scala-lang.org/what-is-scala.html>

Zur Zeit besteht die *Apache Spark*-Architektur aus fünf teilweise voneinander abhängigen Modulen [ASFSpark]:

- *Spark Core*
- *Spark SQL*
- *Spark Streaming*
- *MLlib Machine Learning Bibliothek*
- *GraphX*

Der *Spark Core* ist ein grundlegendes Modul des *Apache Spark*-Systems. Er stellt die Funktionalität der Aufgabenverteilung, der Ablaufplanung, der Ein- und Ausgabeprozesse durch eine Programmierschnittstelle in den Sprachen *Java*, *Scala*, *R* und *Python* bereit. *Spark Core* stellt aus dem physischen Speicher abgelesene Datensätze in der Form von *RDD*-Objekten dar.

Die Komponente *Spark SQL* liegt auf dem *Spark Core* und bietet eine Umwandlung von *RDD*-Objekten in *DataFrames*-Datenabstraktionen, welche *SQL*-Anfragen wie Selektionen, Projektionen, Joins und Gruppierungen an die in *RDD* vorhandenen Datensätze ermöglicht.

Spark Streaming ist eine Erweiterung der *Spark Core*-Programmierschnittstelle, welche eine skalierbare und, mit einem höheren Datendurchsatz, eine fehlertolerante Verarbeitung des Live-Datenstroms ermöglicht. Die Live-Daten können von verschiedenen Strom-Datenquellen wie *Kafka*¹, *Flume*², *Kinesis*³ oder *TCP-Sockets* eingenommen werden und mit Hilfe von komplexen Algorithmen mit der Verwendung von High-Level-Funktionen wie *Map*, *Reduce* und *Join* verarbeitet werden. Schließlich können die verarbeitete Daten in Dateisystemen, Datenbanken und Live-Dashboards herausgegeben werden. Die Komponente funktioniert folgendermaßen: *Spark Streaming* bekommt externe Datenströme und teilt die eingehenden Ströme in Datenblöcke fester Größe auf, welche in *Spark Core* verarbeitet werden. *Spark Streaming* gewährt eine High-Level-Abstraktion: *discretized Stream* oder *DStream*. [SparkStream] *DStream* repräsentiert einen kontinuierlichen Strom von Datenblöcken, welcher als eine Sequenz von *Spark RDDs* dargestellt wird.

Die *MLlib Machine Learning Bibliothek (Spark MLlib)* ist ein *Machine Learning*⁴-Framework, das auf dem *Apache Core* basiert und eine praktische Anwendung des *Maschinellen Lernens* skalierbar und einfach ermöglicht. *Spark MLlib* stellt folgende High-Level-Algorithmen und Werkzeuge zur Verfügung:

- *Machine Learning*-Algorithmen: allgemein bekannte Lernalgorithmen für Klassifizierung, Regression, Clustern und kollaboratives Filtern

1 <https://kafka.apache.org/>

2 <https://flume.apache.org/>

3 <https://aws.amazon.com/kinesis/streams/>

4 http://www.sas.com/en_us/insights/analytics/machine-learning.html

- Featurisierung: Funktionalität der Extraktion, Transformation, Dimensionalitätsreduktion und Selektion
- Pipelines: Werkzeuge für die Konstruierung, Evaluierung und das Tuning von *Machine Learning*-Pipelines¹ (d.h. praktische Werkzeuge für Skalierung und Vereinfachung von Machine Learning)
- Persistenz: Laden und Speichern von Algorithmen, Modellen und Pipelines
- Dienstprogramme: Lineare Algebra, Statistik, Datenverarbeitung

Die Komponente *GraphX* ist eine neue *Spark*-Programmierschnittstelle für Graphen und *graphparallele Berechnungen*² wie zum Beispiel kollaboratives Filtern. *Spark GraphX* erweitert *Spark RDDs* mit *Resilient Distributed Property Graph* – ein gerichteter Multigraph mit Eigenschaften, die an jedem Scheitelpunkt und jeder Kante beigefügt sind. Für die Unterstützung der Graphberechnungen wird in *Spark GraphX* ein Satz von folgenden Grundoperatoren verwendet: *subgraph* – eine Auswahl von Subgraphen, *joinVertices* – eine Zusammenführung von Scheitelpunkten des Graphs, und *aggregateMessages* – eine Aggregation von Nachrichten in Zielscheitelpunkten. Die Komponente bietet auch die optimierte Variante der *Pregel*³ Programmierschnittstelle. *Apache Spark GraphX* beinhaltet eine wachsende Kollektion von Graphalgorithmen und Konstruktionen, welche Aufgaben der Graphanalyse vereinfachen.

Spark lässt sich sowohl auf einem einzigen Server in einem *Standalone-Modus*⁴ als auch auf dem ganzen Cluster mit mehreren Serverknoten installieren. Das *Standalone-Modus* erlaubt Zugriffe auf die Daten durchzuführen, die in den Remotequellen zum Beispiel in *HDFS*, sowie in dem lokalen Filesystem gespeichert sind.

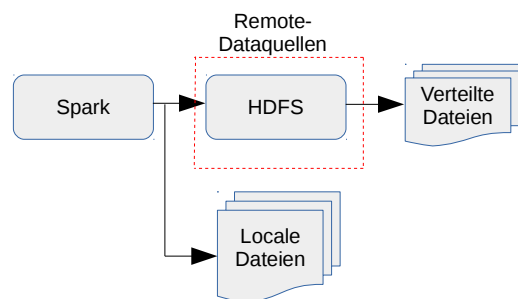


Abbildung 2.4. Nutzung von Datenquellen in *Standalone-Modus* von *Spark*.

Die Abbildung 2.4 präsentiert ein Nutzungsdiagramm verschiedener Datenquellen in einem *Spark*-System, das in einem *Standalone-Modus* installiert wurde. In diesem Fall können

¹ <https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html>

² <http://dl.acm.org/citation.cfm?id=2387883>

³ <https://blog.acolyer.org/2015/05/26/pregel-a-system-for-large-scale-graph-processing/>

⁴ *Standalone-Modus* ist eine Cluster-Computing-Architektur, wobei der Master-Thread und alle Worker-Threads auf der selben Knoten ausgeführt werden [<http://spark.apache.org/docs/latest/spark-standalone.html>].

vom System sowohl lokale als auch die mit Hilfe von *HDFS* im Netzwerk verteilte Dateien verarbeitet werden. Für eine *Cluster-Architektur* von *Apache Spark* ist erforderlich, dass jede Serverknoten eine Zugriffsberechtigung für sämtliche Dateien besitzt. Diese Voraussetzung kann nur bei der Verwendung von Remotedatenquellen erfüllt werden.

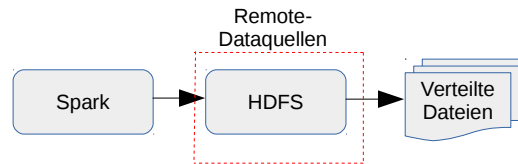


Abbildung 2.5. Nutzung von Datenquellen in *Cluster-Architektur* von *Spark*.

Auf der Abbildung 2.5 wird ein Nutzungsdiagramm für eine *Cluster-Architektur* des *Spark*-Systems vorgestellt. Hier werden als mögliche Datenquellen des entwickelten Systems nur die in *HDFS* gespeicherte Dateien vorgeschlagen, denn eine Anfrage zu den Daten aus einem lokalen Filesystems einer Server-Knote durch eine Zugriffsberechtigung der Dateien verhindert wird.

Im Allgemeinen ist *Apache Spark* ein mächtiges und leistungsvolles Werkzeug für die Big-Data-Verarbeitung.

2.2.3 **SciSpark**

SciSpark ist eine skalierbare wissenschaftliche Verarbeitungsplattform, die interaktive Berechnungen und Explorationen auf der Basis von *Apache Spark* ermöglicht. Dieses Prototypprojekt beschreibt die Architektur der *wissenschaftlichen RDD (scientific RDD: sRDD)*, von Bibliotheken der linearen Algebra und raumbezogenen Operationen. Sein ursprünglicher Fokus liegt auf dem Entwurf des „*Grab em 'Tag em' Graph em' (GTG) Algorithmus*“ in MapReduce Notation. *GTG* ist ein Algorithmus der Merkmal-Erkennung, Merkmal-Weiterentwicklung und Merkmal-Auszeichnung. Der *GTG*-Algorithmus wurde für Klima- und Wetteranwendungen entwickelt, welche mesoskalierte konvektive Komplexe in hoch aufgelösten, zeitlich und raumbezogen Fernerkundungsdatensätzen identifizieren und Merkmale aus verschiedenen Datenquellen beschreiben. Außerdem wird der *GTG*-Algorithmus in der Graphentheorie für die Merkmal-Erkennung und Weiterentwicklung verwendet.[SciSpGit]

Das *SciSpark*-Framework besteht aus zwei Komponenten: dem *Frontend*-Bereich und der *Backend*-Grundlage. In Abbildung 2.6 wird die Architektur der oben genannten Komponenten präsentiert. Der *Frontend*-Bereich präsentiert eine *RESTful*¹ Programmierschnittstelle, welche den Benutzern einen Zugriff auf dieses System über einen Webbrowser und den *HTTP* ermöglicht. Die Abbildung 2.6 zeigt, dass die *Backend*-Grundlage in drei

¹ <https://blogs.msdn.microsoft.com/martinkern/2015/01/05/introduction-to-rest-and-net-web-api/>

Schichten unterteilt wird, welche durch eine Interaktion benutzerdefinierter Berechnungen auf der Basis von *Scientific Resilient Distributed Datasets (sRDD)* mit einander interagieren. Die Funktionalität der Schichten im *SciSpark-Backend* wird im Folgenden beschrieben:

- Das *Persistence Layer (Persistenzschicht)* nimmt wissenschaftliche Dateiformate wie NetCDF und HDF sowohl aus lokalen, als auch aus Remotequellen in einem parallelen Modus an. Das *SciSpark-Framework* liest die Datenquellen mit Hilfe seines „*Uniform Resource Identifiers (URIs)*“ ab, was die Unterschiede der Herkunft von Datensätzen beseitigt. Gleichzeitig wird die *Persistenzschicht* verwendet, um Zwischenergebnisse auszugeben und Endergebnisse zu speichern. Diese Schicht stellt den Datenzugriff für die oberen Schichten bereit und bietet eine Erweiterungsmöglichkeit für verschiedene Datenquellen anderer Modelle.
- Die „*Partition, Extract, Transform and Load (PETaL)*“ Schicht partitioniert und verteilt die in der *Persistenzschicht* vorbereiteten Datenquellen über Rechner-Knoten. Die *PETaL*-Schicht extrahiert die Daten und wandelt sie in einen in *SciSpark* verwendbaren Datentyp um, welcher nach dieser Operation in die *Verarbeitungsschicht* geladen wird.
- Die *Processing-Schicht* ermöglicht eine Ausführung benutzerdefinierter Berechnungen. Diese Berechnungsaufgaben werden mit Hilfe von *sRDD*-Methoden durchgeführt.

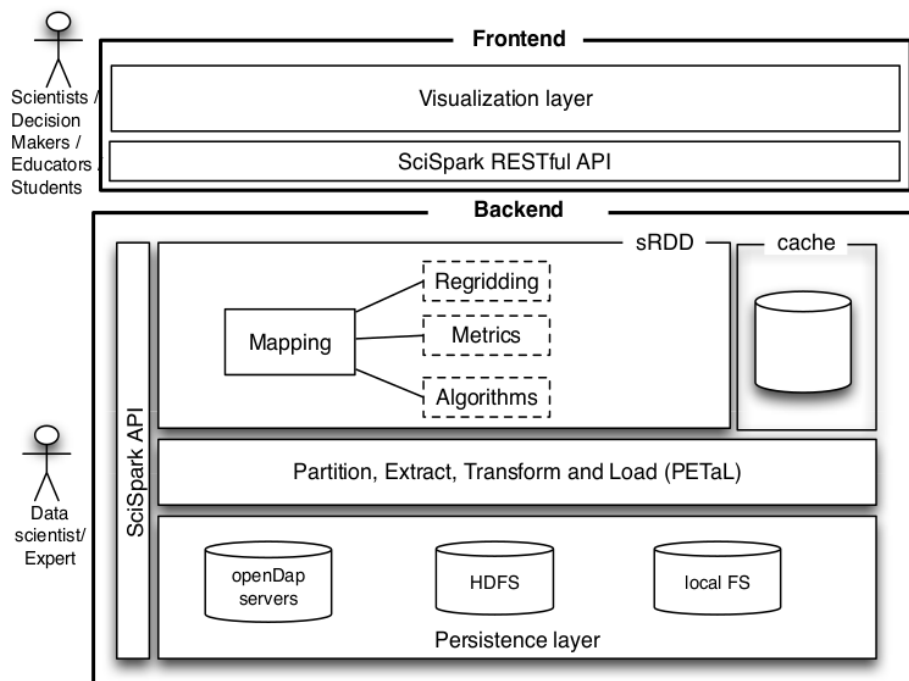


Abbildung 2.6 Die *SciSpark*-Architektur. [SciSparkApp]

Das Konzept des *Scientific Resilient Distributed Dataset (sRDD)* ermöglicht Operationen auf mehrdimensionale Arrays und verteilte *In-Memory-Verarbeitung*. Um dieses Ziel zu erreichen, wurde die Idee der *Selbstdokumentation* in hierarchischen Dateiformaten verwendet, wobei die Dokumentierung in einem *metaData-HashMap*-Objekt gespeichert wird. Auf dieser Basis wurden selbstdokumentierte Klassen mit den Namen *SciTensor* und *SciDataset* bereit gestellt. In folgendem werden die Klassenarchitekturen detailliert erläutert.

2.2.3.1 *SciTensor*

Die *SciTensor*-Architektur stellt eine Logik zur Verfügung, die vorhandene Daten in einem multidimensionalen Format darstellt, welches für leistungsstarke Matrixoperationen die bekannten linearen Algebra-Bibliotheken *ND4J*¹ und *Breeze*² einsetzt. Diese Bibliotheken nutzen Vorteile des lokalen Caches bei elementweiser Verwendung von Operatoren auf dimensional Arrays, welche in physische lineare Arrays zugewiesen werden. Die Abbildung 2.7. präsentiert *sRDD* mit seiner *SciTensor*-Architektur. Hier ist zu sehen, dass das *sRDD*-Objekt aus einem Array von *Partitions* besteht, wobei eine *Partition* auf einem Berechnungsknoten ausgeführt wird. Die *Partitions* beinhalten *SciTensor*-Arrays. Die Architektur des *SciTensor*-Objektes wird mit Hilfe der *Key-Value*-Metadatenstruktur der oben erwähnten *Selbstdokumentation* und einem *AbstractTensor*-Objekt gebildet. Damit können vorhandene NetCDF-Variablen als *AbstractTensor*-Objekte in ein *SciTensor*-Objekt gepackt werden, wobei der Name der Variablen als ein *Key* definiert wird. Die *AbstractTensor*-Objekte sind die Wrappers für *n*-dimensionale Array-Schnittstellen wie *INDArray*³ (auf der Abbildung 2.7 als ein zweidimensionales *AbstractTensor*-Objekt dargestellt), die von den Bibliotheken *ND4J* und *Breeze* als ein Basisdatenformat verwendet werden. Damit erweitert sich die Funktionalität von *sRDD*-Objekten um *ND4J*- und *Breeze*-Operatoren vorhandene.

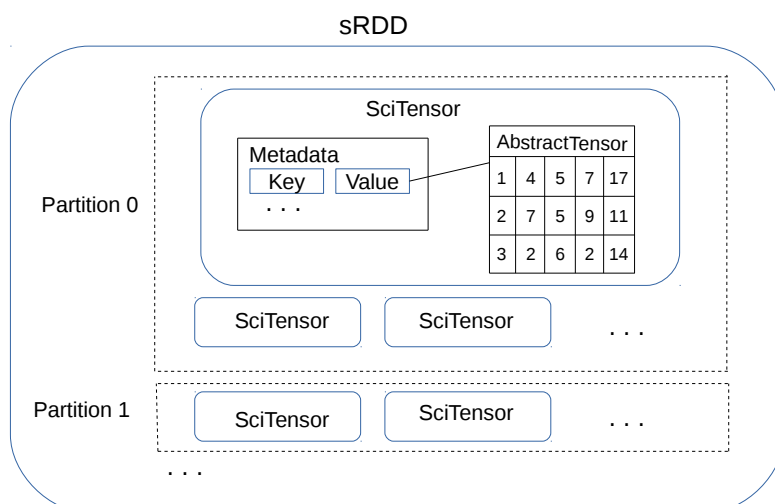


Abbildung 2.7 Die *SciTensor*-Architekturen innerhalb des *sRDD*. [SciSparkApp]

1 <http://nd4j.org/>

2 <http://www.scalanlp.org/api/breeze/>

3 <http://nd4j.org/>

2.2.3.2 SciDataset

Als eine weitere Darstellung von *NetCDF*-Daten wird von *SciSpark* das *SciDataset*-Objekt verwendet. Die Architektur von *SciDataset* wird mit Hilfe der *Key-Value*-Metadatenstruktur für *NetCDF*-Variablen und *NetCDF*-Attribute gebildet. Dabei wird die Funktion der Selbstdokumentation an die *Attributen-Key-Value*-Datenstruktur delegiert.

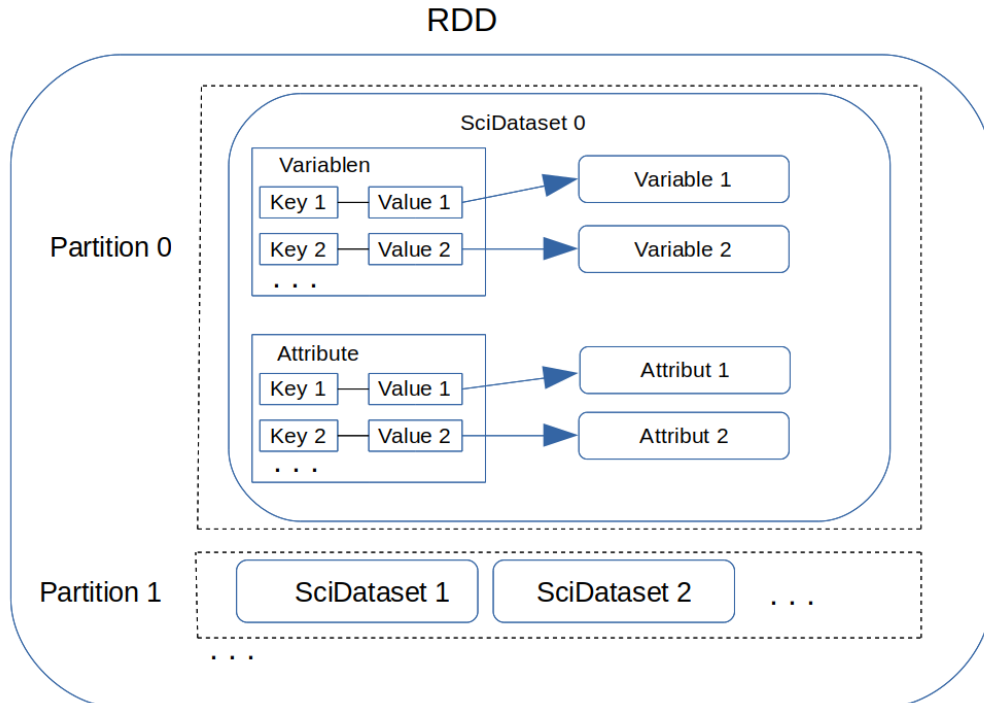


Abbildung 2.8 Die *SciDataset*-Architekturen innerhalb des *RDD*.

Die Abbildung 2.8. präsentiert eine *SciDataset*-Architektur innerhalb des *RDD*-Objektes. Hier ist zu sehen, dass das *RDD*-Objekt aus einem Array von *Partitions* besteht. Jede *Partition* beinhalten Daten aus einer *NetCDF*-Datei. Die Architektur des *SciDatasets* besteht aus zwei *LinkedHashMap*-Objekten, wobei ein *LinkedHashMap*-Objekt für die Beschreibung von *NetCDF*-Variablen dient und zweites beinhaltet globale Attribute der *NetCDF*-Datei. Als *Key*-Wert in den *LinkedHashMap*-Objekten wird der Name des Objektes in textueller Form verwendet. Als *Value*-Werte in den Attributen-*HashMap* werden textuelle Beschreibungen von den *NetCDF*-Attributen eingesetzt. Die *Value*-Werte des Variablen-*HashMaps* werden in *Variable*-Objekten gespeichert. Diese Objekte beschreiben alle Datenfelder von *NetCDF*-Variablen und sind lokale Wrapper in *SciSpark*-System.

Auflistung 2.6 präsentiert ein Beispiel des Ablesens von *NetCDF*-Dateien in das *RDD*-Array der *SciDataset*-Objekten, wobei nach einer Initialisierung des *SciSparkContextes* (Zeile 2) einer Eingang von *NetCDF*-Daten folgt (Zeile 3). Die eingehende *NetCDF*-Dateien werden mit Hilfe einer *netcdfWholeDatasets*-Funktion abgelesen. Die Parametern dieser Funktion sind eine Textdatei mit Pfaden zu den *NetCDF*-Dateien (Zeile 4), eine Liste mit angeforderten *NetCDF*-Variablen (Zeile 5), und eine Zahl der Partitionierung für die Datenverarbeitung (Zeile 6). In diesem Beispiel wird die *netcdfWholeDatasets*-Funktion

verwendet, die ein *RDD*-Array (nicht *sRDD*) mit *SciDataset*-Objekten zurückgibt. In der aktuellen *SciSpark*-Version fehlt die Implementierung entsprechender Funktionalität für *sRDD*-Arrays.

```
1 ...
2 val sc = new SciSparkContext(new SparkConf())
3 val sTensor:RDD[SciTensor] = sc.netcdfWholeDatasets(
4     "/path/to/list_of_files.txt",
5     List("time", "longitude", "latitude", "sf"),
6     2)
7 ...
```

Auflistung 2.6. Auslesen von NetCDF-Daten in das *RDD*-Array von *SciDataset*-Objekten.

2.2.3.3 Zusammenfassung

Das *SciSpark*-Projekt stellt den Entwicklern mit einer klaren Architektur eine Möglichkeit für die Weiterentwicklung neuer Methoden der Partitionierung, Extraktion, Transformation und des Ladens von Daten aus verschiedenen Formaten bereit. Innerhalb von *SciSpark* können hoch aufgelöste Gitter mit Hilfe von komplexeren sequenziellen Algorithmen ohne Einschränkungen bei den Matrix-Größen verarbeitet werden. Die *SciSpark*-Architektur vereint Vorteile von verteilten und sequenziellen Programmierungen bei Softwarelösungen benutzerdefinierter Aufgaben.

2.2.4 Apache Zeppelin

Apache Zeppelin ist ein *Notebook* für mehrere Zwecke interaktiver Datenanalyse. *Apache Zeppelin* ist eine webbasierte Mehrzweckumgebung, die es Dateningenieuren, Datenanalysten und Wissenschaftlern ermöglicht, bei der Entwicklung, dem Organisieren, der Ausführung, der Verteilung des Berechnungscodes und der Visualisierung der Ergebnisse ohne einen Verweis auf die Kommandozeile und ohne Kenntnisse benötigter Clusterdetails produktiver zu werden. Durch eine Speicherung von benutzten Programmcodes, Verarbeitungsergebnissen und Grafikdiagrammen in einem *Notebook*-Object können die geleisteten Arbeiten aufbewahrt und erneut ausgeführt werden. *Notebooks* erlauben den Benutzern sowohl eine interaktive Ausführung als auch eine Speicherung von komplizierten Workflows. Das *Notebook* ermöglicht die Erstellung von schönen datengetriebenen, interaktiven und kollaborativen Dokumenten, die mit *SQL*, *Scala*, *Java* erzeugt werden können. *Apache Zeppelin* stellt folgende Datenoperationen zur Verfügung [AZIntr]:

- *Datenaufnahme*
- *Datenidentifikation (Data Discovery)*
- *Datenanalyse*
- *Datenvisualisierung und Daten-Kooperation (Data Collaboration)*

Zur Zeit unterstützt *Apache Zeppelin* ein mehrsprachiges Backend, welches für ein wachsendes Ökosystem von Datenquellen zuständig ist. Durch eine Interpretierung der Snippet-Kodes aus dem Frontend-Notebook werden die Befehle zu den entsprechenden Backend-Systemen weitergeleitet, welche die eingegebenen Aufgaben verarbeiten. Damit stellt *Apache Zeppelin* den Datenwissenschaftlern ein interaktives „*snippet-at-time*“ Verfahren zur Verfügung. Derzeit unterstützt *Apache Zeppelin* in der Basisversion folgende Interpreten: *Apache Spark*, *Python*¹, *JDBC*², *Apache Cassandra*³, *Appache Kylin*⁴ und die Unix Kommandozeile. Durch eine Erweiterung der Einstellungen kann diese Liste vergrößert werden. Die Abbildung 2.9 stellt die Architektur von *Zeppelin* dar, wobei sich Zugriffe zu dem *Zeppelin*-Server über folgende zwei Wege realisieren lassen:

1. REST-Schnittstelle: für einen Aufbau der Verbindung von drittseitigen Systemen
2. Weboberfläche: für eine bequeme und produktive Benutzerinteraktion.

Auf dem *Zeppelin*-Server werden ausgeführte Notebook-Abläufe über gewählte Interpreten an die entsprechenden oben erwähnten Frameworks weitergeleitet.

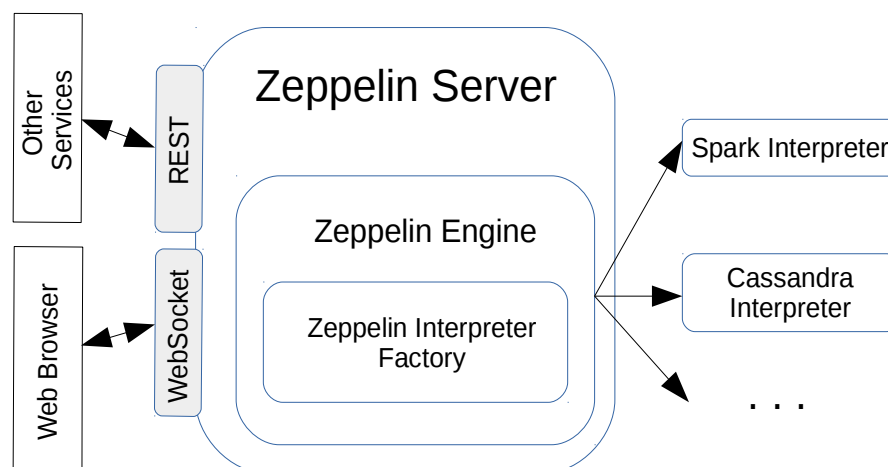


Abbildung 2.9. Apache Zeppelin Architektur. [ZepVis]

Eine besondere Option des *Apache Zeppelin* ist ein integrierter Interpret des *Apache Spark*-Frameworks. Die *Apache Zeppelin*-Umgebung mit der *Spark*-Integration bietet:

- Automatisch erstellte *SparkContext*- und *SQLContext*-Objekte

¹ *Python* ist eine universelle Programmiersprache, die mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen wurde [https://www.python.org/]

² *JDBC (Java Database Connectivity)* ist eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller auf der Basis von Java-Plattform [http://www.oracle.com/technetwork/java/javase/jdbc/index.html].

³ *Apache Cassandra* ist ein einfaches, verteiltes Datenbankverwaltungssystem für sehr große strukturierte Datenbanken [http://cassandra.apache.org/]

⁴ *Apache Kylin* ist ein verteiltes Analyse-System, welches SQL-Schnittstellen und mehrdimensionale Analysen der extrem großen ausgelegten in Hadoop Datenmengen zur Verfügung stellt. [http://kylin.apache.org/].

- Eine Laufzeitladung von *Jar*¹-Abhängigkeiten aus der lokalen Datenumgebung und aus den *Maven-Repositories*²
- Möglichkeiten des *Job-Abbruchs* und der *Progressvisualisierung* (visueller Indikator des Zustands von dem Progress)

Apache Zeppelin bietet ein integriertes Werkzeug für gewöhnliche Diagramme. Die Datenvisualisierung ist nicht nur auf *SparkSQL*-Abfragen beschränkt, sondern jede Ausgabe von einer beliebigen unterstützten Programmiersprache kann erkannt und visualisiert werden. *Apache Zeppelin* stellt dynamische Webformulare für eine Eingabe von Parametern zur Verfügung. Als ein Vorteil dieses Systems lässt sich eine *Notebook-URL* von mehreren Datenwissenschaftlern gemeinsam nutzen und wendet alle Befehlsänderungen in Echtzeit an. Insgesamt ist *Apache Zeppelin* ein bequemes Werkzeug, um die Datenverarbeitung einfacher und schneller durchzuführen. [AZIntr]

2.2.5 Jupyter

Das *Jupyter*-Notebook ist eine Webanwendung, die eine Erstellung und eine Verteilung von Dokumenten ermöglicht, welche ausführbare Programmcodes, Gleichungen, visuelle Darstellungen und Erläuterungstext beinhalten. [JN]

Das *Jython*³-Projekt wurde in 2014 gestartet und wird seitdem als eine *Client-Server*-Anwendung weiterentwickelt. Die Anwendung besteht aus zwei grundlegenden Komponenten: *Kernels* und einem *Dashboard*. Ein *Kernel (Kern)* ist das Programm, welches die Benutzerkodes prüft und ausführt. Die *Jupyter*-Anwendung beinhaltet immer einen Kern für Python-Kodes und beliebige Kerne für andere Programmiersprachen. Das *Dashboard* der Anwendung ist eine Benutzerwebschnittstelle, welche nicht nur für die Erstellung, Korrektur, Öffnung und eine Ausführung von *Jupyter-Notebook*-Dokumenten dient, sondern auch die vorhandenen Kerne verwaltet. Mit Hilfe von *Dashboard* werden benötigte Kerne eingeschaltet und unnötige Kerne ausgeschaltet.

Jupyter umfasst folgende Bereiche der Big-Data-Analyse: Datenbereinigung, Datentransformationen, numerische Simulation⁴, statistische Modellierung, maschinelles Lernen. Das Notebook unterstützt mehr als vierzig Programmiersprachen einschließlich der populärsten in der Datenwissenschaft: *Python*, *R*, *Julia*⁵ und *Scala*. Die Webanwendung

1 *Jar (Java ARchive)* ist ein Packet-Dateiformat für die Zusammenführung von *Java*-Klassen, zugehörigen Metadaten und Ressourcen (Bilder usw.) in einem Datei für die weiteren Verteilung [http://docs.oracle.com/javase/6/docs/technotes/guides/jar/index.html].

2 *Maven Repository* wird verwendet, um Build-Artefakte und Abhängigkeiten von verschiedenen Typen zu halten [https://maven.apache.org/guides/introduction/introduction-to-repositories.html].

3 *Jython* ist eine Java-Implementierung der Programmiersprache Python und ermöglicht somit die Ausführung von Python-Programmen auf jeder Java-Plattform [https://jython.org/].

4 *Direkter Numerischer Simulation* ist die rechnerische Lösung der vollständigen instationären Navier-Stokes-Gleichungen [https://en.wikipedia.org/wiki/Direct_numerical_simulation].

5 *Julia* ist eine höhere Programmiersprache, welche für numerisches und wissenschaftliches Rechnen entwickelt wurde [http://julialang.org/].

stellt reichhaltige Ausgabemöglichkeiten wie Bilder, Videos, JavaScripts und LaTeX¹-Kodes zur Verfügung, welche bei der Manipulierung und Visualisierung von Daten in einem Echtzeit-Modus verwendet werden. Durch eine Zusammenarbeit mit Cluster-Computing-Frameworks wie *Apache Spark*, *Apache Hadoop* erweitert *Jupyter* seine Leistungen im Datenverarbeitungsbereich. Mit Hilfe der Integration solcher Bibliotheken wie *pandas*², *scikit-learn*³, *dplyr*⁴ stellt das Notebook verschiedene Funktionen aus dem maschinellen Lern-Bereich, der Statistik und der Big-Data-Analyse bereit.

Das *Jupyter-Notebook* bietet den Benutzern verschiedene Arten der Authentifizierung. In dem System werden folgende Modelle unterstützt: *Pluggable Authentication Modules*⁵-*System (PAM)*, *OAuth*⁶. *Jupyter-Notebook* ermöglicht eine Zusammenarbeit mit anderen Systemen über das Zugriffsmodell von linuxbasierten Systemen.

2.2.6 OPeNDAP

OPeNDAP ist eine Datentransport-Architektur mit eigenem Protokoll, welches auf der Basis von *Data Access Protocol*⁷ (*DAP*) aufgebaut wurde, was eine „name-type-value“-Anfrage an die Nutzdaten ermöglicht. Das Akronym „*OPeNDAP*“ beschreibt „*Open-source Project for a Network Data Access Protocol*“ [*OpenDAP*], welches von *OPeNDAP.org*⁸ entwickelt wird.

Die *OPeNDAP*-Architektur basiert auf dem *Client-Server*-Modell. In dem einfachsten Zugriffsmodell der Interaktion zwischen *Client* und *Server* werden von einer *Client*-Anwendung beliebige Datenanfragen an einen *Server* geschickt, welcher mit entsprechenden Daten antwortet.

Als ein *Client* für den *OPeNDAP-Server* kann auch ein Internetbrowser benutzt werden. Beim Verwenden von Webbrowsern entstehen einige Beschränkungen auf die Datenmengen. Eine Zielgruppe der Verwendung der *OPeNDAP*-Clientskomponente sind verschiedene Webdienste, die in der eigenen Funktionalität verteilte Datenquellen benötigen.

¹ *LaTeX* ist ein Softwarepaket, das die Benutzung des Textsatzsystems TeX mit Hilfe von Makros vereinfacht [<https://www.latex-project.org/>].

² *Pandas* ist eine Python-Bibliothek für die Datenmanipulationen und Datenanalyse [<http://pandas.pydata.org/>].

³ *Scikit-learn* ist eine Python-Bibliothek des maschinellen Lernens [<http://scikit-learn.org/stable/>].

⁴ *Dplyr* ist R-Bibliothek für die Datenmanipulation auf der Basis von Split-Apply-Combine-Methodologie [<https://github.com/hadley/dplyr>].

⁵ *Pluggable Authentication Modules (PAM)* ist eine Programmierschnittstelle der Benutzerauthentifizierung mit Hilfe konfigurierbarer Module. [https://en.wikipedia.org/wiki/Pluggable_authentication_module].

⁶ *OAuth* ist ein offenes Protokoll, das eine standardisierte, sichere API-Autorisierung für Desktop-, Web- und Mobile-Anwendungen erlaubt. [<https://de.wikipedia.org/wiki/Oauth>].

⁷ *Data Access Protocol* ist ein Datenübertragungsprotokoll, das speziell für wissenschaftliche Daten entwickelt wurde [<https://earthdata.nasa.gov/standards/data-access-protocol-2>].

⁸ *OPeNDAP* ist eine Datentransportarchitektur und ein Protokoll, das in der Geowissenschaft weit verbreitet ist [<https://www.opendap.org/>].

Die *OPeNDAP*-Gruppe stellt eine Implementierung des Standardserverprotokolls zur Verfügung. Der *OPeNDAP*-Server letzter Generation wird „*Hyrax*“ genannt. Er hat ein modulares Design, welches für die Bereitstellung der öffentlich zugänglichen Client-Schnittstellen ein leichtes Java-Servlet und einen Backend-Daemon „*Back End Server*“ (*BES*) für die leistungsanspruchsvolle Verarbeitung verwendet.

Die *OPeNDAP.org* definiert folgende Vorteile der Nutzung des *Hyrax*-Systems:

- Die Servlet-Architektur ist schneller, robuster und sicherer als *CGI*¹-aufgerufene Perl-Skripte.
- Eine einzelne Installation ist datenformatunabhängig und kann mehrere Datendarstellungen (HDF4, HDF5, NetCDF, etc.) bedienen
- Funktionalität des *THREDDS-Katalogs*² (*Thematic Realtime Environmental Distributed Data Services*)
- Eine *SOAP*³-Schnittstelle für *OPeNDAP*-Datendienste.

Die *OPeNDAP*-Architektur erstellt kein zentrales Archiv von Daten. Die Daten werden in *OPeNDAP* auf ähnliche Weise wie im *World Wide Web*⁴ organisiert. Deswegen wird für die *SOAP* (ursprünglich für Simple Object Access Protocol) ist ein Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls durchgeführt werden können Verfügbarkeit der Daten nur eine Konfigurierung des Servers zu den Netzwerkknoten mit entsprechenden Daten benötigt.

2.2.7 CDO: Climate Data Operators

Climate Data Operators (CDO) ist eine Kollektion von Operatoren, die für die Standardverarbeitung von Klima- und Prognosemodelldaten dient. *CDO* wird durch das Max-Planck-Institut für Meteorologie entwickelt und ist in der Klimaforschung weit verbreitet. [CDO16]

Die Operatoren beinhalten einfache arithmetische und statistische Funktionen, Datenauswahl- und Unterabtastungswerkzeuge sowie räumliche Interpolation, welche *GRIB*⁵- (*General Regularly-distributed Information in Binary form*) und *NetCDF-Dateiformate* unterstützen müssen.

¹ *Common Gateway Interface (CGI)* ist ein Standard für den Datenaustausch zwischen einem Webserver und dritter Software, die Anfragen bearbeitet. [https://de.wikipedia.org/wiki/Common_Gateway_Interface].

² *THREDDS-Katalogs* ist Middleware, um die Lücke zwischen Datenanbietern und Datenbenutzern zu überbrücken. [<http://www.unidata.ucar.edu/software/thredds/current/tds/catalog/>].

³ *Simple Object Access Protocol (SOAP)* ist ein Netzwerkprotokoll für den Datenaustausch zwischen verteilten Systemen und die Ausführung von Remote Procedure Calls [<https://www.w3.org/TR/soap/>].

⁴ *World Wide Web* ist ein über das Internet abrufbares System von elektronischen Hypertext-Dokumenten [https://de.wikipedia.org/wiki/World_Wide_Web].

⁵ *GRIB* ist ein standardisiertes, komprimiertes binäres Datenformat, das üblicherweise in der Meteorologie verwendet wird [https://www.dwd.de/DE/derdwd/it/_functions/Teasergroup/grib_de.html]

Die wichtigsten Eigenschaften von CDO sind:

- Reiche Funktionalität, zu diesem Zeitpunkt gibt es über 700 Operatoren.
- Modulare Struktur und leichte Erweiterbarkeit mit neuen Funktionen.
- Einfache UNIX-Befehlszeilenschnittstelle.
- Ein Datensatz kann von mehreren Operatoren verarbeitet werden, ohne die Zwischenergebnisse in Dateien zu speichern. Dieses Verfahren wird als eine *Pipeline*-Ausführung bezeichnet (Auflistung 2.3.).
- Die meisten Operatoren behandeln unvollständige Datensätze (mit fehlenden Werten).
- Schnelle Verarbeitung großer Datensätze.
- Unterstützung vieler verschiedener Grid-Typen.
- Getestet auf vielen UNIX- und Linux-Systemen, Cygwin² und MacOS-X.

Die Operatoren lassen sich durch den Kommandozeilenbefehl *cdo* ausführen, welcher mit mehreren Parametern für die entsprechend ausgewählten Funktionen befolgt wird. Dabei werden Ein- und Ausgangsdateien als Parameter in die Operatoren übergeben. Die Auflistung 2.5 beschreibt beispielsweise mehrere Einzelbefehle mit der Speicherung von Zwischenergebnissen im Filesystem.

```
1 cdo selvar,sf input.nc tmp1.nc # Selektiert nur den Datensatz
                                von der „sf“-Variable
2 cdo monmean tmp1.nc tmp2.nc   # Kalkuliert Monatsmittelwerte
3 cdo meravg tmp3.nc tmp4.nc    # Kalkuliert meridionale
                                Durchschnittswerte
4 cdo zonavg tmp4.nc output.nc  # Kalkuliert zonale
                                Durchschnittswerte
```

Auflistung 2.5. CDO-Einzelbefehle.

Mit Hilfe der *Pipeline*-Ausführung kann der Ablauf von Befehlen aus der Auflistung 2.5 in eine *Pipeline*-Befehlszeile realisiert werden. Dieses Verfahren wird in der Auflistung 2.6 dargestellt. Hierbei werden mehrere Prozesse generiert, einer pro Operator und über eine Pipeline verbunden.

```
cdo zonavg -meravg -monmean -selvar,sf input.nc output.nc
```

Auflistung 2.6. CDO Pipeline-Ausführung.

² *Cygwin* ist eine Kompatibilitätsschicht, die die Unix-API für verschiedene Versionen von Windows zur Verfügung stellt, [<https://www.cygwin.com/>].

CDO hat auch einige Einschränkungen für *GRIB*- und *NetCDF*-Dateien. Eine *GRIB*-Datei muss, ähnlich wie *NetCDF*, konsistent sein. Das bedeutet, dass innerhalb eines Zeitschritts jede Variable nur einmal auftreten kann und alle Zeitschritte dieselben Variablen haben müssen. Die *NetCDF*-Dateien werden nur für klassische Datenmodelle und Arrays bis zu vier Dimensionen unterstützt. Diese Abmessungen sollten nur für horizontale und vertikale Grids und für die Zeitmessungen verwendet werden. Die *NetCDF*-Attribute sollten den GDT¹-, COARDS²- oder CF-Konventionen³ folgen. [CDO16]

2.2.8 Stencil Algorithmen

Bevor der Begriff *Stencil Algorithmus* erläutert wird, müssen einige Definitionen gegeben werden. Zunächst wird der Begriff *Stencil Code* erklärt. Die *Stencil Codes* (Schablonencodes) sind eine Klasse von *iterativen Kerneln*⁴, die Array-Elemente nach einem festen Muster, genannt Schablone, aktualisieren. Dies ist typischerweise für numerische Algorithmen notwendig, aber auch bei der Datenanalyse. [StencilWiki] Im Umfang dieser Arbeit werden Abläufe von drei Mustern für ein-, zwei- und dreidimensionalen Arrays vorgestellt. Im folgendem wird einen Ablauf mit einem eindimensionalen Array mit n Elementen präsentiert. In dem Ablauf dieses Arrays wird für jedes Element E mit dem Index x einen neuen Wert nach der Funktion $E_x = f(E_{x-1}, E_{x+1})$ berechnet. Für ein zweidimensionales Array (quadratisches Gitter) wird klassisches Ablaufmuster der *Von-Neumann-Nachbarschaft*⁵ angewendet (Abbildung 2.8), wobei die Nachbarelemente eine rechnerische Auswirkung auf das zentrale Element des Gitters haben. Die Abbildung 2.8 präsentiert zweidimensionale *Von-Neumann-Nachbarschaft* mit *Manhattan-Distanz*⁶ von 1. Hier wird für beliebiges Element P mit dem Index (x,y) einen neuen Wert nach der Funktion $P_{x,y} = f(D_{x-1,y}, D_{x+1,y}, D_{x,y-1}, D_{x,y+1})$ berechnet.

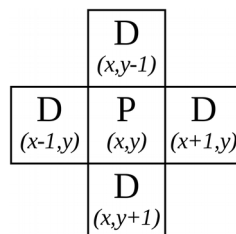


Abbildung 2.8. Zweidimensionale *Von-Neumann-Nachbarschaft* mit *Manhattan-Distanz* von 1. [VonNeumann]

1 GDT ist eine Reihe von Vereinbarungen für den Austausch und die gemeinsame Nutzung von Dateien, die auf der Basis der NetCDF API erstellt wurden [http://www-pcmdi.llnl.gov/drach/GDT_convention.html].

2 COARDS ist eine Konventionen zur Standardisierung von NetCDF-Dateien (1995) [http://ferret.wrc.noaa.gov/noaa_coop/coop_cdf_profile.html].

3 Die *Climate and Forecast (CF) Konvention* ist ein Standard für Metadaten der NetCDF-Dateien [<http://cfconventions.org/>].

4 *Iterativen Kernel* (in der Programmierung) sind Körper von Schleifen.

5 Die *Von-Neumann-Nachbarschaft* ist eine Nachbarschaftsbeziehung in einem quadratischen Raster (benannt nach John von Neumann) [<https://de.wikipedia.org/wiki/Von-Neumann-Nachbarschaft>].

6 *Manhattan-Distanz* ist eine Form der Geometrie, wobei die übliche Distanzfunktion oder Metrik der euklidischen Geometrie durch eine neue Metrik ersetzt wird, in der der Abstand zwischen zwei Punkten als eine Summe der absoluten Differenzen ihrer kartesischen Koordinaten ist [https://en.wikipedia.org/wiki/Taxicab_geometry].

Für die Erklärung des dreidimensionalen Modells der *Von-Neumann-Nachbarschaft* wird die Abbildung 2.9 vorgestellt. Wie zu sehen ist, kommt zu dem zweidimensionalen Modell eine zusätzliche Dimension mit zwei beeinflussenden Bestandteilen. Die Einflüsse werden auf der Abbildung 2.9 mit Hilfe von roten Pfeilen gezeigt. Die Berechnungsfunktion für zweidimensionale Arrays kann für dritte Dimension erweitert werden:

$$P_{x,y,z} = f(D_{x-1,y,z}, D_{x+1,y,z}, D_{x,y-1,z}, D_{x,y+1,z}, D_{x,y,z+1}, D_{x,y,z-1}).$$

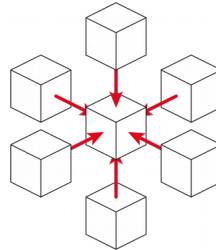


Abbildung 2.9. Dreidimensionale *Von-Neumann-Nachbarschaft* mit *Manhattan-Distanz* von 1. [StencilWiki]

Unter *Stencil Algorithmus* wird ein automatisiertes Anwendungsmodell von dem Stencil Code auf dem N -Dimensionalen Array verstanden, wobei für den *iterativen Kernel* eine benutzerdefinierte Funktion mit bestimmten dimensionalen Einflüsselementen verwendet wird.

Die *Stencil Algorithmen* verwendet man oft in Programmcodes im Rahmen von wissenschaftlichen und technischen Anwendungen der Rechnersimulation beispielsweise in der *numerischen Strömungsmechanik*¹, Klimaforschung und Bildverarbeitung.

Zusammenfassung

In diesem Kapitel wurden wissenschaftliche Dateiformate und Verarbeitungstechnologien vorgestellt. Das NetCDF-Datenformat und das SciSpark-Big-Data-Framework wurden detailliert erläutert. Außerdem sind andere verwandte Komponenten und Technologien der Datenbewahrung und der Datenverarbeitung kurz vorgestellt worden, sodass die theoretische Basis dieses Kapitels eine detaillierte Erläuterung vom Aufbau der Webanwendung im weiteren Verlauf ermöglicht.

¹ Die *numerische Strömungsmechanik* ist eine etablierte Methode der Strömungsmechanik [https://de.wikipedia.org/wiki/Numerische_Strömungsmechanik].

3 Design

Dieses Kapitel beschreibt die Anforderungen an ein grundlegendes Modell und die Funktionalität des Analyse-Systems. In dem Abschnitt 3.1 wird ein genauerer Aufbau des Systems präsentiert. Die verwendeten Datenstrukturen und Datenverwaltung werden in dem Abschnitt 3.2 vorgestellt. Zum Schluss werden einige Abläufe des Analyse-Systems in dem Abschnitt 3.3 dargelegt.

3.1 Methodik

Für eine Ermöglichung der Analyseprozessen von Klimadaten in dem NetCDF-Dateiformat werden in dem entwickelten Big-Data-Werkzeug Vorgänge verwendet, die auf den Basis von *Climate Data Operators* und *Stencil Algorithmen* entwickelt werden und aus einer interaktiven Umgebung steuerbar und ausführbar sind. Daraus lassen sich folgende technisch angeforderte Komponenten des Systems ergeben:

- Ein *leistungsvolles Computing-Framework* für die effiziente Datenverarbeitung (*Backend*)
- Eine *interaktive Benutzeroberfläche* für die Verwaltung und Darstellung von Datenprozessen (*Frontend*)
- Die *Datenquelle* für Bereitstellung von Eingabedaten und Speicherung von Ausgabedaten in *NetCDF-Dateiformat*

Im Folgenden werden die oben erwähnte Komponenten vorgestellt. Dabei wird in Abschnitt 3.1.1 eine Begründung zur Auswahl eines passenden Computing-Frameworks gegeben. Abschnitt 3.1.2 erläutert die Verwendung einer interaktiven Benutzeroberfläche. In Abschnitt 3.1.3 sind Datenquellen des entwickelten System beschrieben. Das grundlegende Modell des Systems wird in Abschnitt 3.1.4 präsentiert. Schließlich wird in Abschnitt 3.1.5 die Entscheidung zur Auswahl passender Programmiersprachen für die Entwicklung der Funktionalität von *Frontend*- und *Backend*-Komponenten verdeutlicht.

3.1.1 Auswahl des Cluster-Computing-Frameworks

In dem zweiten Kapitel dieser Arbeit wurden zwei Cluster-Computing-Frameworks: *Apache Spark* mit einer Erweiterung für wissenschaftliche Zwecke *SciSpark* und *Apache Hadoop* präsentiert. Diese Frameworks werden als potenzielle Computing-Kerne des entwickelten Systems betrachtet.

Als wesentliche Anforderung für das Computing-Framework definiert sich das Berechnungspotential des Systems. Deswegen werden Leistungsfähigkeiten von *Apache Hadoop* und *Apache Spark* auf der Basis experimenteller Tests miteinander verglichen. Dafür wurde eine einfache Berechnungsaufgabe ausgedacht, in der die in zweiten Kapitel erwähnte theoretische Aspekte über die Systemleistungen von beiden Frameworks auch praktisch bekräftigt werden können. Diese Aufgabe besteht aus Berechnungen von funktionalen Werten der *logistischen Regression*¹ für jedes aufkommendes Wort aus einem literarischen Werk von Johann Wolfgang von Goethe „Faust“. Diese Aufgabe wird während des Berechnungsprozesses auf zwei Teile untergliedert:

- *Kalkulieren von Worten.* In diesem Teil wird berechnet, wie oft jedes Wort in dem literarischen Werksinhalt auftritt.
- *Berechnung der logistischen Regression für jedes Wort.* Hier wird für jedes Wort, auf dem Grund seiner Frequenz, der Wert seiner *logistischen Regression* berechnet. Die Werte *logistischer Regression* werden mit Hilfe der folgenden Formel berechnet:

$$f(x) = \frac{1}{1 + e^{-x}}$$

wobei X ist die Frequenz des Wortes

Durchläufe des Experiments werden auf folgenden Versionen der Software realisiert:

Apache Spark 2.0.0 für *Hadoop 2.7*

Apache Hadoop 2.7.3

Diese Systeme werden in dem *Single-Node-Cluster-Modus*² getestet.

Für die Durchführung des Experiments wird ein handelsüblicher Laptop mit folgender Architektur verwendet:

| | |
|----------------------|--|
| Modell: | Lenovo S400 |
| CPU: | Intel(R) Core(TM) i5-3317U CPU @ 1.70GHz |
| Arbeitsspeicher: | 8 GByte |
| Physischer Speicher: | SSD KINGSTON SV300S37A 240GByte |
| Betriebssystem: | Ubuntu 16.10 x86_64 |
| Swap: | 9.5 GByte |

¹ https://de.wikipedia.org/wiki/Logistische_Regression

² <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleNode.html>

Um die Ergebnisse des Experiments von möglichst vielen nebenläufigen Einflüssen des Systems zu schützen, wird jeder Berechnungsablauf zehn mal wiederholt. Aus den Gründen der Besonderheit von den Frameworksarchitektur können diese Wiederholungen nicht in einer Schleife realisiert, denn in diesem Fall einige Zustände des Berechnungsablaufs in dem Arbeitsspeicher gecached werden. Das *Caching* kann großen Einfluss auf die Richtigkeit des Experiments haben. Deswegen wurden die Wiederholungen mit Hilfe von einem *Shell*-Script ausgeführt.

Während des Ablaufs von dem Experiment wurden die beiden Big-Data-Frameworks mit folgenden Iterationsmengen getestet: 1, 5, 10, 15 und 20. Die Iterationsmengen so wie die implementierte *logistische Regression*-Algorithmen dienen für eine steuerbare Belastung der Systemen, um die Leistungsfähigkeiten von beiden Systemen analysieren zu können. Die Experimentergebnisse sind in Abbildung 3.1 vorgestellt. Am Diagramm sieht man, dass mit der Erhöhung von Iterationen in der Ablaufschleife der Unterschied der Verarbeitungszeit zwischen diesen Systemen rasant steigt. Bei der höchsten Stufe der Iterationsmenge mit zwanzig Abläufen der Schleife weichen die Verarbeitungszeiten zwischen *Spark* und *Hadoop MapReduce* mehr als in fünfundzwanzig mal ab. Dadurch wurde experimentell bestätigt, dass *Apache Spark* sogar bei zwanzig Iterationszyklen deutlich leistungsfähiger als *Hadoop MapReduce* ist.

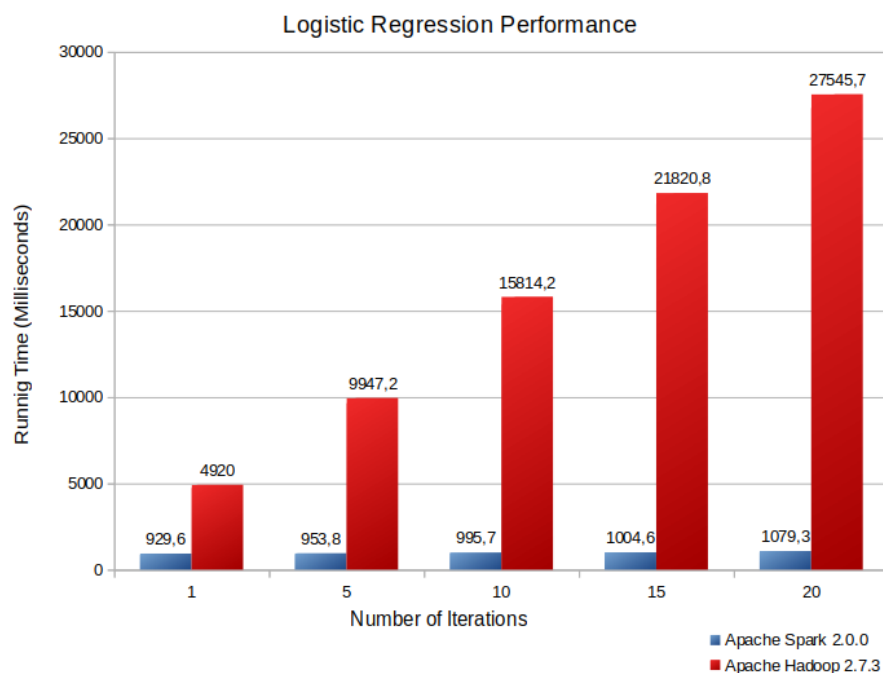


Abbildung 3.1. Leistungsfähigkeit von *Spark* und *Hadoop MapReduce*.

Auf dem Grund des Berechnungspotentials von beiden Systemen ergab sich die Entscheidung zur Auswahl von *Apache Spark* als eine Basis für einen Aufbau von Cluster-Computing-Framework. Daneben werden in dem entwickelten System Komponente des in Abschnitt 2.2.3 beschriebenen *SciSpark*-Framework verwendet.

3.1.2 Auswahl der interaktiven Benutzeroberfläche

Als interaktiver Benutzeroberfläche wurde webbasierte interaktive Umgebungen genommen, die eine Plattformunabhängigkeit und leichte Zugänglichkeit durch die Verwendung von Webbrowsern für die Benutzer erschafft.

Im zweiten Kapitell sind zwei webbrowserbasierte interaktive Umgebungen: *Apache Zeppelin* und *Jupyter* beschrieben, die mit den *Apache* Big-Data-Frameworks integrierbar sind. In der Tabelle 3.1 werden wesentliche Aspekte und deren Bewertungen für die beiden Umgebungen vorgestellt.

| Aspekte | <i>Apache Zeppelin</i> | <i>Jupyter</i> |
|---|-------------------------|--------------------------------------|
| Integration mit Apache Spark | +++ (native integriert) | + (integrierbar durch Konfiguration) |
| Sprachumwandlung zu der Programmiersprache des Cluster-Computing-Frameworks | +++ | + |
| Interaktive Webschnittstelle | +++ | +++ |
| Erstellung und Ausführung von Notebook-Dokumenten | +++ | +++ |
| Vorintegrierten Kerne | ++ | +++ |
| Leicht konfigurierbar | ++ | ++ |
| Datenvisualisierung: 3D Charts | +++ | +++ |
| Datenvisualisierung: 3D Surface | + | ++ |
| Documentation | + | + |
| Community-Aktivität | + | + |
| Erlernbarkeit | ++ | ++ |
| Preis | +++ (kostenlos) | +++ (kostenlos) |
| Persönliche Erfahrung | ++ | + |

Tabelle 3.1. Aspekte der Auswahl der Weboberfläche.

Auf der Basis der Vergleichstabelle von *Apache Zeppelin* und *Jupyter* ist eine Entscheidung getroffen worden, die *Apache Zeppelin*-Umgebung zu der interaktiven Benutzerweboberfläche des entwickelten Systems auszuwählen.

3.1.3 Datenquellen

Das entwickelte System auf der Basis von *Apache Spark* und *SciSpark*-Framework verwendet als Datenquellen sowohl lokal gespeicherte als auch verteilte *NetCDF*-Dateien mit Hilfe von *Hadoop Distributed File System (HDFS)* und *OPeNDAP* im Netzwerk.

3.1.4 Grundlegende Architektur

Auf dem Grund der technischen Anforderungen, die am Anfang dieses Kapitels formuliert wurden, wird die grundlegende Architektur des entwickelten Systems im Folgenden zusammengefasst. In oben erwähnten Abschnitten dieses Kapitels wurden die technischen Entscheidungen gerechtfertigt, die für den Aufbau des entwickelten Systems angewendet werden. Abbildung 3.2 präsentiert die Architektur dieses System mit der Verwendung ausgewählter Komponenten, wobei als interaktive Benutzerschnittstelle das *Zeppelin*-Framework hingelegt wird, für die Datenverarbeitungen auf einem Cluster das *SciSpark*-System verwendet wird, welches auf die *NetCDF*-Dateien aus *HDFS*, *OPeNDAP* und lokalem Filesystem zugreifen kann. Das *Zeppelin*-Framework ist auch für die grafische Darstellungen der Ergebnisse von Datenverarbeitung zuständig.

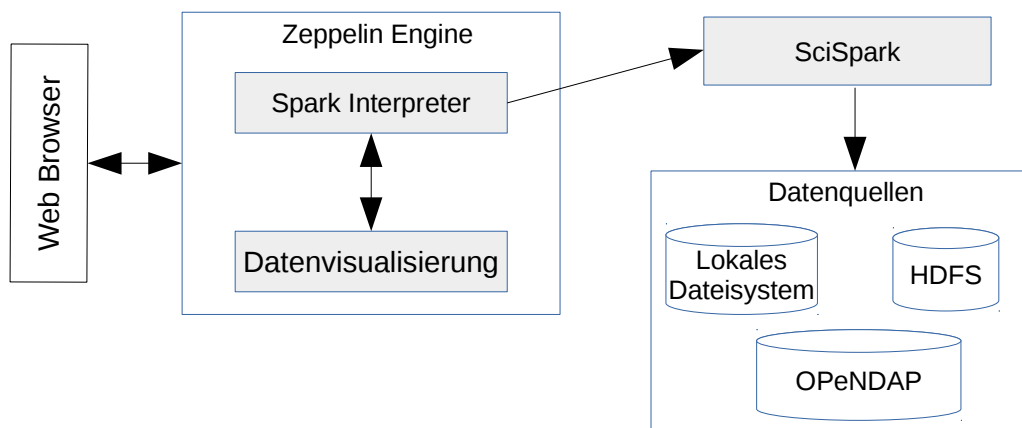


Abbildung 3.2. Architektur des entwickelten Systems.

3.1.5 Auswahl der Programmiersprachen und Bibliotheken

Dieser Abschnitt erläutert grundlegende Faktoren der Auswahl von Programmiersprachen für die *Cluster-Computing*- und Datenvisualisierungskomponenten. Dafür werden in Abschnitt 3.1.5.1 Vor- und Nachteile unterstützter Programmiersprachen für *Cluster-Computing-System* abgewogen. In Abschnitt 3.1.5.2 werden mögliche Programmiersprachen für die Datenvisualisierung verglichen.

3.1.5.1 Programmiersprache für die Cluster-Computing-Komponente

In dem Abschnitt 3.1.4 ist entschieden worden, als eine *Cluster-Computing*-Komponente das *SciSpark*-Framework zu verwenden. Das *SciSpark*-Framework als eine Ableitung von *Apache Spark* bietet eine Vielfalt von Programmierschnittstellen in den Programmiersprachen: *Java*, *Scala*, *R* und *Python* zur Auswahl. Deswegen ist die Wahl einer Programmiersprache für *Apache Spark* beziehungsweise für *SciSpark* eine Angelegenheit, welche auf einer Grundlage der Effizienz von funktionalen Lösungen basiert. Im Folgenden werden einige grundlegende Faktoren für die wichtigsten Programmiersprachen von Programmierschnittstellen des *Spark*-Frameworks untersucht.

Die *R*-Programmiersprache wird auf Grund von Leistungsberechnungen aus der Auswahlliste beseitigt. Eine interaktive Datenanalyse in *R* ist begrenzt, da die *R*-Laufzeit in einem *Single-Threading* ausgeführt wird, welcher nur Datensätze verarbeiten kann, die in den Speicher einer einzelnen Berechnungsmaschine passen können. Deswegen wird es in *Spark* die Anbindung an *R* durch eine Brücke zur *Java*-Programmierschnittstelle gelöst, was folgende negative Wirkungen auf die Systemleistung hat:

- Durch eine Datenumwandlung erhöht sich die Wahrscheinlichkeit von dem Auftritt von Fehlern
- Ein Leistungsverlust tritt durch Umwandlung auf

Bei der Eignungsprüfung von *Java* für die Verwendung in *Spark* wurden folgende Mängel nachgewiesen, die entsprechend diese Programmiersprache aus der Vergleichsliste ausgeschlossen:

- *Java* ist im Vergleich zu anderen Schnittstellensprachen zu ausführlich (der Code in *Java* länger als z. B. In *Scala*).
- Der *Read-Evaluate-Print-Loop* (REPL)¹ wird in *Spark* für *Java*- Programmiersprache nicht unterstützt, was bei der Auswahl einer Programmiersprache für große Datenverarbeitung ein entscheidender Punkt ist.

Nun bleiben in der Auswahlliste noch *Scala* und *Python*, welche für die Entwicklung der Datenverarbeitungskomponente des Systems benutzt werden könnten. Die beiden Programmiersprachen haben prägnante Syntax, reiche Funktionalität. Beide sind objektorientiert. Es gibt auch wichtige Punkte, wo *Scala* die *Python*-Programmiersprache übertrifft. Im Allgemeinen ist *Python* langsamer als *Scala*². Im Ziel dieser Arbeit besteht eine Notwendigkeit zur Entwicklung eigener Funktionalität einer erheblichen Verarbeitungs-

¹ Der *Read-Eval-Print Loop* (REPL) ist als *interaktives Toplevel* oder *Shell Sprache* bekannt. *REPL* ist eine einfache, interaktive Computerprogrammierungsumgebung, die einzelne Benutzereingaben (Ausdrücke) ausführt, auswertet und die Ergebnisse an den Benutzer zurückgibt [https://en.wikipedia.org/wiki/Read-eval-print_loop].

² *Scala* vs. *Python*: Vergleichstabelle [<http://vschart.de/vergleich/scala/vs/python-programming-language>]

logik, die in ausgewählter Programmiersprache implementiert wird. Für diesen Zweck bietet *Scala* eine bessere Leistung. Ein weiterer Vorteil von *Scala* ist das, dass *Spark* in der *Scala*-Programmiersprache geschrieben wurde. Wenn von dem *Python*-Wrapper die zugrunde liegenden *Spark*-Kodes, die in *Scala* geschrieben wurden, auf einer *JVM (Java Virtual Machine)* aufgerufen werden, können die Quellcodes einer Umwandlung zwischen zwei verschiedenen Umgebungen und Programmiersprachen eventuell zu Problemen führen. Außerdem ermöglicht die Verwendung von *Scala* einen unverzüglichen Zugriff auf die neusten Aktualisierungen der *Spark*-Funktionalität. Auf der Basis von obenerwähnten Gründen wird *Scala*-Programmiersprache für die Entwicklung neuer Funktionalität von der Datenverarbeitungskomponente verwendet.

3.1.5.2 Programmiersprache für die Datenvisualisierung

Als eine Benutzerweboberfläche bietet *Apache Zeppelin* reiche Interaktionsmöglichkeiten für die Prozesssteuerung und für die Präsentation von Ergebnisse, welche sich in einem *HTML-CSS-JavaScript*-Verbund realisiert. Grundlegendes Ziel der Erweiterung von der *Zeppelin*-Standardwebschnittstelle ist eine grafische Visualisierung von dreidimensionalen Datensätze mit Hilfe von dreidimensionalen Oberflächendiagrammen. Auf Grund der Struktur des entwickelten Systems können oben genannte Oberflächendiagramme sowohl in Berechnungskonten (*Backend*) als auch im Webbrowser (*Frontend*) generiert werden. Eine Generierung von Diagrammen im *Backend* lässt sich mit Hilfe von entsprechenden Bibliotheken in den Programmiersprachen *Python* und *R* ausführen. Damit wird in das *Frontend*-Bereich ein fertiges Bild des dreidimensionalen Oberflächendiagramms übertragen, welches für Benutzer keine Interaktionsmöglichkeit über die Daten zulässt. Im Gegenteil zum *Backend*-Bereich, stellt die Erzeugung des Oberflächendiagramms im *Frontend*-Bereich mit Hilfe einer *JavaScript*-Bibliothek ein reiches Interaktionsmodell zur Verfügung, was dem Benutzer eine besondere Flexibilität der Diagrammanalyse bietet. Deswegen ist eine Entscheidung getroffen, die Generierung von Oberflächendiagrammen in das *Frontend*-Bereich zu verschieben. Zum heutigen Zeitpunkt bietet *Apache Zeppelin* keine Option mit den beinhalteten Werkzeugen und Bibliotheken eine dreidimensionale grafische Auswertung im *Frontend* durchzuführen. Deswegen wird eine externe *JavaScript*-Bibliothek in *Zeppelin* integriert. Die Auswahl eine *JavaScript*-bibliothek sowie die Auswahlkriterien werden in dem kommenden Abschnitt 3.1.5.3 erläutert.

3.1.5.3 Auswahl der JavaScript-Bibliothek

Das entwickelte System realisiert die Visualisierung der verarbeiteten Daten durch die Funktionalität einer *JavaScript*-Bibliothek, die in dem *Frontend* ausgeführt wird. Um eine Unterstützung der bestmöglichen Interaktivität zur Verfügung zu stellen, werden für die

Entwicklung dieses Systems die neusten Webtechnologien wie *HTML5*¹- und *CSS3*²-Standards eingesetzt. In diesem Abschnitt werden einige *JavaScript*-Bibliotheken, die sich mit *Apache Zeppelin* integrieren lassen, in verschiedenen Aspekten verglichen. In der Tabelle 3.2. werden *JavaScript*-Frameworks: *AngularJS*³, *Data-Driven Documents (D3.js)*⁴ und *Plotly.js*⁵ miteinander verglichen.

| Aspekte | <i>AngularJS</i> , | <i>Data-Driven Documents (D3.js)</i> | <i>Plotly.js</i> |
|------------------------------|--------------------|--------------------------------------|------------------|
| Kompatibilität mit Zeppelin | +++ | + | + |
| Größe | +++ | ++ | + |
| Einfache Datenvisualisierung | ++ | ++ | +++ |
| 3D Charts | +++ | +++ | +++ |
| 3D Surface | + | ++ | +++ |
| Dokumentation | + | + | ++ |
| Community-Aktivität | +++ | ++ | ++ |
| Erlernbarkeit | + | + | ++ |
| Preis | +++ | +++ | +++ |
| Persönliche Erfahrung | + | + | ++ |

Tabelle 3.2. Aspekte der Auswahl der JavaScript-Bibliothek.

Das *Plotly.js*-Framework stellt mit einer Größe von ca. 1,7 MiB das größte *JavaScript*-Framework. Beim Einsetzen einer Bibliothek mit solcher Größe in gewöhnliche Webanwendungen, kann dies ein Ausschlußkriterium sein. Die Besonderheit der Größe von zwei Megabyte kann für wissenschaftliche Webanwendung für Analysen von Datenmengen in Größen von Hunderten Megabyte vernachlässigt werden.

Alle oben genannte Frameworks bieten Kompatibilität mit der *Apache Zeppelin* Mehrzweckumgebung. Im Vergleich zu anderen Bibliotheken befreit *AngularJS* den Entwickler von Integrierungsproblemen und hat die geringste Größe. Gleichzeitig stellt es beschränkte Funktionalität für die dreidimensionale grafische Darstellung zur Verfügung. Bei der Auswahl der *AngularJS*-Bibliothek muss die Integration zu *Apache Zeppelin* implementiert werden.

1 <https://www.w3.org/TR/html5/>

2 <https://www.w3.org/TR/2014/REC-css-namespaces-3-20140320/>

3 <https://angularjs.org/>

4 <https://d3js.org/>

5 <https://plot.ly/javascript/>

Wesentliche Kriterien bei der Frameworkauswahl sind die einfachere Verwendung der Anwendungsprogrammierschnittstelle und gute Nutzungsdokumentation der Bibliotheken. Die Datenvisualisierung soll möglichst leicht implementierbar sein. Nach der Berücksichtigung aller Faktoren praktischer Nutzung wurde eine Entscheidung zugunsten des *Plotly.js*-Frameworks getroffen.

3.2 Datenstrukturen und Datenverarbeitung

Bei der Entwicklung des Systems für die Analyse von Klimadaten werden Datenstrukturen verwendet, die auf die *Frontend*- und der *Backend*-Komponenten des Systems verteilt werden. Für das entwickelte System wurde als eingehendes Datenformat die *NetCDF*-Dateien definiert. Mittels vorhandener *SciSpark*-Funktionalität der *Backend*-Komponente können die Dateien in zwei Strukturklassen: *SciDataset* und *SciTensor* abgelesen werden, welche sich mit Hilfe von *RDD*-Arrays im Laufe der Verarbeitungsprozessen über die Cluster-Knoten verteilen können. Im Laufe der Systementwicklung wurde ein neues *CdoDataset*-Objekt kreiert, welches eine Struktur hat, die ähnlich zu dem *SciDataset*-Objekt ist. Das *CdoDataset*-Objekt erweitert die vorhandene Funktionalität der *SciDataset*-Objekten mit *CDOs*- und *Stencil*-Merkmale. Als Ausgangsdaten werden im System *NetCDF*-Dateien und grafische Darstellungen in *JPEG*¹- und *PNG*²-Formaten angeboten, wobei sich die Datenvisualisierung in der *Frontend*-Komponente mit Hilfe einer *JavaScript*-Bibliothek ausführen lässt, welche als einen Standard für komplexe Datenobjekte *JavaScript Object Notation (JSON)*³-Format verwendet. Deswegen werden im *Frontend* für die Datenanalyse und grafische Darstellung von Daten sämtliche Datenstrukturen in ein *JSON*-Format und *JavaScript*-Arrays konvertiert. Eine Umwandlung von sämtlichen Datenformaten in ein *JSON*-Objekt wird im *Backend*-Bereich durchgeführt. Im Folgenden werden sowohl die Datenstrukturen vom *Backend* wie *CdoDataset*-Objekt und von im verwendenden Subobjekten, als auch die *JSON*-Datenstruktur für die *Frontend*-Komponente vorgestellt.

3.2.1 *CdoDataset* Datenstruktur

Die Struktur von einem *CdoDataset*-Objekt beschreibt den kompletten Inhalt einer *NetCDF*-Datei mit Hilfe von verlinkten *HashMap*-Kollektionen. Die Auflistung 3.1 präsentiert die Struktur eines *CdoDataset*-Objektes in *Scala*-Programmiersprache. Dieser Datenstruktur organisiert vorhandenen in der *NetCDF*-Datei Variablen als eine *HashMap*-Kollektion mit *Key-Value*-Paaren, wobei *Key*-Werte der Kollektion die Namen von Variablen in einem *String*-Format beschreiben und *Value*-Werte die *Variablen*-Objekte beinhalten. Die globale Attribute einer *NetCDF*-Datei lassen sich genauso in einer *HashMap*-Kollektion speichern, wobei *Key*- und *Value*-Werte in diesem Fall als *String*-

1 <https://en.wikipedia.org/wiki/JPEG>

2 https://en.wikipedia.org/wiki/Portable_Network_Graphics

3 <http://www.json.org/>

Objekte beschrieben werden. Das *datasetName*-Feld (Zeile 4) bezeichnet in dieser Struktur eine textuelle Beschreibung von dem Pfad zu der *NetCDF*-Datei und dem Dateiname.

```
1 class CdoDataset(  
2     val variables: mutable.HashMap[String, Variable],  
3     val attributes: mutable.HashMap[String, String],  
4     var datasetName: String) extends Serializable {  
6     ...  
7 }
```

Auflistung 3.1. Datenstruktur von *CdoDataset*-Objekt.

Eine Erläuterung des *Variable*-Objektes von *SciSpark* wird in Abschnitt 3.2.2 gegeben.

3.2.2 Variable Datenstruktur

Das *Variable*-Objekt von *SciSpark* wird auf der Auflistung 3.2 vorgestellt. In dem Objekt lassen sich der Variablenname (Zeile 2 der Auflistung 3.2) und der Datentyp (Zeile 3) von den beinhalteten Datensätzen durch entsprechende *String*-Felder dieses Objektes beschreiben. In dem Feld *array* (Zeile 4) werden Nutzdaten der Variable in einem *AbstractTensor*-Objekt vorgestellt. Die *AbstractTensor*-Struktur wird als eine Abstraktion von mehrdimensionalen Arrays dargelegt. Durch die Verwendung dieser Abstraktion vererben Datensätze des *Variable*-Objektes sämtliche Funktionalität aus den *INDArray*-Schnittstellen von *ND4J*¹ und *Breeze*² Bibliotheken, die die erwähnte Abstraktion für die Implementierung eigener Merkmale benutzen. So wie in den *CdoDataset*-Objekten werden die Variablenattribute in einem Textform von *Key-Value*-Kollektion gespeichert (Zeile 5). Das wesentliche Feld für die Variablen-datenstruktur ist *dims* (Zeile 6). Mit Hilfe dieses Feldes wird eine Dimensionalitätsliste der Nutzdaten von der Variable vorgestellt. Als Elemente der Liste werden *Name-Dimensionswert*-Paaren verwendet, wobei der *Name* eine textuelle Beschreibung der Dimension vorliegt und der *Dimensionswert* entsprechende Dimensionalität definiert.

```
1 class Variable(  
2     var name: String,  
3     val dataType: String,  
4     val array: AbstractTensor,  
5     val attributes: mutable.HashMap[String, String],  
6     val dims: List[(String, Int)]) extends Serializable {  
7     ...  
8 }
```

Auflistung 3.2. Datenstruktur des *Variable*-Objektes.

¹ <http://nd4j.org/>

² <http://www.scalanlp.org/api/breeze/>

3.2.3 JSON-Struktur der Datenvisualisierung

Durch einen Umwandlungsprozess generiert das entwickelte System im *Backend*-Komponente aus einem *CdoDataset*-Objekt eine *JSON*-Datenstruktur. Auf der Auflistung 3.3 wird eine *JSON*-Object präsentiert, die für eine Darstellung von einem Oberflächendiagramm für das Frontend zur Verfügung gestellt wird. Hier sieht man, dass die *JSON*-Struktur die Arrays verschiedener Dimensionalität in sich beinhaltet. Beispielsweise haben *x* und *y* Elemente (Zeilen 2 und 3) des *data_tensor*-Objektes eine Dimension, während das Element *z* (Zeilen 4 bis 8) ein zweidimensionales Array ist. Das präsentierte *JSON*-Objekt kann auch komplexeren Elemente, beispielsweise *colorbar* (Zeile 10 der Auflistung 3.3), beinhalten.

```
1 var data_tensor =[{
2   x:[60.0, 59.25, 58.5, ... , 51.0, 50.25],
3   y:[30.0, 30.75, 31.5, ... , 39.0, 39.75],
4   z:[ [0.00006122, 0.00008852, 0.00008852, ... ,0.00304203],
5       [0.00007229, 0.00009368, 0.00010475, ... ,0.00390581],
6       ...
7       [0.00012466, 0.00078116, 0.00139783, ... ,0.00136537]
8   ],
9   type:'surface',
10  colorbar:{
11      autotick: true,
12      title: 'm of water equivalent'
13  }
14 }];
```

Auflistung 3.3. *JSON*-Datenstruktur für die Datenvisualisierung im Frontend.

3.2.4 Datenverarbeitung mit CDOs

Im Umfang dieser Arbeit werden *CDO*-Funktionalität, *Stencil*-Algorithmus und Visualisierungswerkzeug für das entwickelte System implementiert. Aus der enormen Mengen der vorhandenen *CDO*-Operatoren folgt, dass eine komplette Implementierung aus zeitlichen Gründen im Laufe dieser Arbeit nicht realisierbar ist. Deswegen werden die Operatoren nach funktionalen Aspekten klassifiziert und einige Operatoren aus den Klassifikationsgruppen in das *SciSpark*-Framework übertragen. Nach der Klassifizierung der *CDOs* ergaben sich folgende Gruppen: *Informationsoperatoren*, *Fileoperatoren*, *Selektionsoperatoren*, *Vergleichmerkmale*, *Einstellungsoperatoren*, *mathematische* und *statistische* Operatoren, Operatoren der *Korrelation* und *Interpolation*, Operatoren *spektraler Transformation*, sowie *Import*- und *Export*-Operatoren. Eine Implementierung einiger Operatoren wird aus technischen und zeitlichen Gründen ausgeschlossen. Beispielsweise die Gruppe der *Einstellungsoperatoren* speziell *setctomiss* (Merkmal für die Setzung einer Konstante für die fehlenden Werte) ist technisch in den *SciSpark*-Datenstrukturen nicht realisierbar, denn vorhandene Datenstrukturen *SciDataset* als auch *SciTensor* unterstützen keine fehlenden Werte. Die Nutzdaten dieser Datenstrukturen werden

in einem *AbstractTensor* gespeichert, wobei eine Verwendung von undefinierten Elemente komplett ausgeschlossen ist.

Die in dem entwickelten System implementierten *CDO*-Operatoren werden mit einer kurzen Beschreibung in folgender Liste dargestellt:

➤ Informationsoperatoren:

- *showName* - die Funktion zeigt Namen der Variablen, die in der gegebenen *NetCDF*-Datei vorhanden sind.
- *showYear*-Operator präsentiert alle Jahre der Zeitdimension in einem „yyyy“-Format.
- *showMonth* legt alle Monate der Zeitdimension in einem „MM“-Format dar.
- *showDate* gibt eine Datumsliste von der Zeitspanne in einem „yyyy-MM-dd“-Format zurück.
- *showTime* zeigt alle Zeitstempeln in einem „HH:mm:ss“-Format, die in der Zeitspanne vorhanden sind.
- *showTimestamp* präsentiert alle Zeitstempeln in einem „yyyy-MM-dd'THH:mm:ss“-Format, die in der *NetCDF*-Datei vorhanden sind.

➤ Selektionsoperatoren:

- *selTime* filtert Datensätze sämtlicher Variablen nach Zeitstempeln von Zeitdimensionen in einem „hh:mm:ss“-Format aus. Alle Datensätze, die außerhalb dieser Zeitspanne liegen, werden weggelassen.
- *selVar* selektiert die Datensätze einer verlangten Variable mit dem Namen *variableName*, gegebenen als Parameter, und liefert Ergebnisse in einem *CdoDataset*-Objekt zurück.
- *selDay* filtert Datensätze sämtlicher Variablen nach Zeitstempeln in einem „dd“-Format. Bei der Anwendung dieser Funktion können sowohl kommagetrennte Liste von Daten in einem „dd, dd, ..., dd“-Format als auch Zeitspannen in einem „dd – dd“-Format benutzt werden. Alle Datensätze, die außerhalb dieser Zeitspanne oder Liste liegen, werden weggelassen.
- *selMonth* filtert Datensätze sämtlicher Variablen nach Zeitstempeln in einem „MM“-Format. Bei der Anwendung dieser Funktion können sowohl kommagetrennte Liste von Monaten in einem „MM, MM, ..., MM“-Format als auch Zeitspannen in einem „MM – MM“-Format benutzt werden. Alle

Datensätze, die außerhalb dieser Zeitspanne oder dieser Liste liegen, werden weggelassen.

- *selYear* filtert Datensätze sämtlicher Variablen nach Zeitstempeln in einem „yyyy“-Format. Bei der Anwendung dieser Funktion können sowohl kommagetrennte Liste von Jahren in einem „yyyy, yyyy, ..., yyyy“-Format als auch Zeitspannen in einem „yyyy – yyyy“-Format benutzt werden. Alle Datensätze, die außerhalb dieser Zeitspanne oder dieser Liste liegen, werden weggelassen.
- *selLonLatBox* filtert Datensätze sämtlicher Variablen nach einem Intervall von Längengraden, welches zwischen den minimalen und maximalen Werten der Gradn liegt, und einem Intervall von Breitengraden, das zwischen den minimalen und maximalen Werten dieser Gradn sich befindet. Als Parametern für diese Funktion werden Werte für die Box übergeben: *startLongitude*, *endLongitude*, *startLatitude*, *endLatitude*.

➤ Statistische Operatoren:

- *merMean* berechnet Durchschnittswerte für jede Variable über die Breitengrade, als das Ergebnis einer Variable wird für jeden Längengrad der Durchschnittswert aller Breitengrade ausgegeben.
- *merMin* ermittelt minimale Werte für jede Variable über die Breitengrade (für jeden Längengrad wird der minimale Wert aller Breitengrade ausgegeben).
- *merMax* rechnet maximale Werte für jede Variable über die Breitengrade (für jeden Längengrad wird der maximale Wert aller Breitengrade ausgegeben).
- *merAvg* kalkuliert Mittelwerte für jede Variable über die Breitengrade (für jeden Längengrad wird der Mittelwert aller Breitengrade ausgegeben).
- *merStd* berechnet Standardabweichungen für jede Variable über die Breitengrade (für jeden Längengrad wird die Standardabweichung aller Breitengrade ausgegeben).
- *merSum* ermittelt Summenwerte für jede Variable über die Breitengrade (für jeden Längengrad wird die Summe aller Breitengrade ausgegeben).
- *merVar* rechnet Varianzwerte für jede Variable über die Breitengrade (für jeden Längengrad wird der Varianzwert aller Breitengrade ausgegeben).

- *zonMean* kalkuliert Durchschnittswerte für jede Variable über die Längengrade, als das Ergebnis einer Variable wird für jeden Breitengrad der Durchschnittswert aller Längengrade ausgegeben.
- *zonMin* ermittelt minimale Werte für jede Variable über die Längengrade (für jeden Breitengrad wird der minimale Wert aller Längengrade ausgegeben).
- *zonMax* rechnet maximale Werte für jede Variable über die Längengrade (für jeden Breitengrad wird der maximale Wert aller Längengrade ausgegeben).
- *zonAvg* kalkuliert Mittelwerte für jede Variable über die Längengrade (für jeden Breitengrad wird der Mittelwert aller Längengrade ausgegeben).
- *zonStd* berechnet Standardabweichungen für jede Variable über die Längengrade (für jeden Breitengrad wird die Standardabweichung aller Längengrade ausgegeben).
- *zonSum* ermittelt Summenwerte für jede Variable über die Längengrade (für jeden Breitengrad wird die Summe aller Längengrade ausgegeben).
- *zonVar* rechnet Varianzwerte für jede Variable über die Längengrade (für jeden Breitengrad wird der Varianzwert aller Längengrade ausgegeben).
- *monMean* kalkuliert Durchschnitte der Werten vorhandener Variablen für jeden Monat über die ganze Zeitspanne.

Mit einer Verwendung der Kombinationen von oben geschrieben *CDO*-Merkmalen kann die Funktionalität von mehreren anderen Operatoren dieses Werkzeuges abgedeckt werden.

Für eine Datenvisualisierung in dem entwickelten System wird eine Funktion zur Verfügung gestellt, die sämtlichen Datensätze dreidimensionaler Variablen als ein Oberflächen-diagramm präsentiert. Die Syntax dieser Funktion ist:

plot3D(varName)

Als ein Parameter wird hier eine Textvariable übergeben, die einen Name von *NetCDF*-Variable in der bearbeiteten Datei beschreibt. Die ausgewählte Variable muss dreidimensional sein.

3.2.5 Datenverarbeitung mit Stencil

Mit Hilfe von *Stencil*-Algorithmen und benutzerdefinierter Aktualisierungsfunktion werden Modellierungen von Klimaprozessen sowie Interpolationen von Daten ermöglicht. Im Umfang dieser Arbeit wurde ein Ablauf über die eins- zwei- und dreidimensionalen *NetCDF*-Daten für *Stencil*-Algorithmus entwickelt, wobei sich die Aktualisierungsfunktion von Benutzern des Systems definieren lässt. Hierbei wurden in dem entwickelten System entsprechende Abstraktionsschnittstellen realisiert: *IStencil1DPattern*, *IStencil2DPattern*, *IStencil3DPattern*, welche die Abläufe von *Stencil*-Verfahren für beliebige Variablen durchführen. Jede Schnittstelle definiert eine Aktualisierungsfunktion *applyFunction*, die einen neuen Wert für das aktuelle Element des Variablendatensatzes berechnet. Das aktuelle Element wird aus der Schleife über Variablenwerte in die *applyFunction* innerhalb einer der folgenden *CdoDataset*-Merkmale übergeben: *stencil1DPattern*, *stencil2DPattern* und *stencil3DPattern*. Eine detaillierte Erklärung zu der Implementierung wird im vierten Kapitell gegeben.

3.3 Ablauf

In diesem Abschnitt werden Systemabläufe innerhalb der entwickelten Anwendung vorgestellt, wobei Abläufe der Datenanfragen an verschiedene Datenquellen erläutert werden, wie ein Wissenschaftler mit dem System Arbeiten und Ausgehende Information kriegen kann. Zum Schluss wird ein Beispielablauf präsentiert, der auf der Basis eingehender Datensätzen grundlegende Abläufe der Datenanalyse und Datenvisualisierung zeigt.

3.3.1 Datenanfrage an eine lokale Datenquelle

Eine Datenanfrage an eine lokale Datenquelle lässt sich mit zwei Verfahren ermöglichen. Als erstes Verfahren kann man eine direkte Datenanfrage an eine vorhandene *NetCDF*-Datei definieren, welche in dem lokalen Filesystem gespeichert ist. Dafür wird ein Verfahren angewendet, welches als ein Funktionsparameter einen direkten Pfad zu einer bestimmten *NetCDF*-Datei darlegt. Damit lässt sich im *Backend*-Bereich eine *CdoDataset*-Struktur mit sämtlichen Daten aus der *NetCDF*-Datei erzeugen.

Bei der zweiten Technik wird die Datenanfrage mit Hilfe von einer Textdatei organisiert, die in dem eigenen Inhalt eins oder mehrere Pfade zu den *NetCDF*-Dateien beschreibt. Mit dieser Vermittlung von der Hilfsdatei wird vom System eine parallele Verarbeitung von diesen Dateien organisiert. In diesem Fall wird in dem *Backend*-Bereich ein *RDD*-Array von

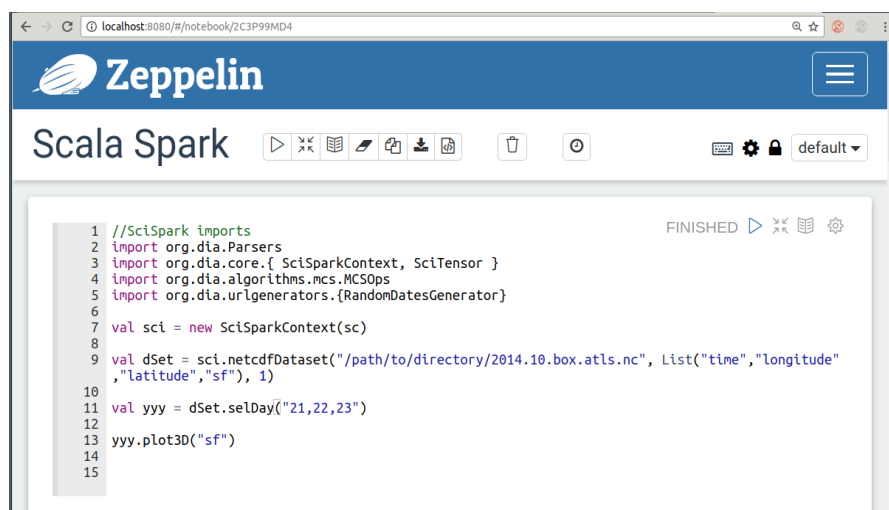
CdoDataset-Objekten erzeugt, wobei die *RDD*-Struktur dafür sorgt dass die *CdoDataset*-Objekte parallel verarbeitet werden.

3.3.2 Datenanfrage an eine Remotedatenquelle

Im Wesentlichen lässt sich eine Datenanfrage an die Remotedatenquellen ähnlich zu den Anfragen an die lokalen Dateien gestalten. Ein einziger Unterschied ist, dass bei den Anfragen an die lokalen Datenquellen ein vollständiger Dateipfad benutzt wird. Im Fall mit den Remotedatenquellen werden stattdessen *Uniform Resource Identifiers*¹ (*URI*) verwendet. Wie es in dem Abschnitt 3.1.3 erläutert wurde, werden unter den Remotedatenquellen innerhalb dieser Arbeit die *HDFS*- und *OpeNDAP*-Systeme verstanden. Die beiden verteilte Dateisysteme unterstützen dieses Verfahren. Beim Eingehen von Dateien aus diesen verteilten Dateisystemen werden entsprechende *RDD*-Arrays mit einzelnen *CdoDataset*-Objekten erstellt.

3.3.3 Ablauf der Datenverarbeitung

Für die Erläuterung von Abläufen der Datenverarbeitung werden einige Anfragen an die Datenquellen erfordert. Dabei erfolgt eine interaktive Eingabe von sämtlichen Parametern, die von den Funktionen der Datenanfragen verlangt werden. Auf der Abbildung 3.3 wird der Beispielablauf der Datenverarbeitung und Visualisierung der Ergebnisse in der Benutzeroberfläche des *Apache Zeppelin* präsentiert. Hier wird nach erster Initialisierung von dem *SciSparkContext*-Objekt (Zeile 7) eine Datenanfrage an die vom Benutzer ausgewählter *NetCDF*-Datei mit dem Namen *2014.10.box.atls.nc* erstellt (Zeile 9). Dabei wird eine Liste mit angefragten Variablennamen und einer Zahl der zurückgegebenen Partitionen von Datensätzen (Zeile 9) mitübertragen.



```
1 //SciSpark imports
2 import org.dta.Parsers
3 import org.dta.core.{ SciSparkContext, SciTensor }
4 import org.dta.algorithms.ncs.MCSOps
5 import org.dta.urlgenerators.{RandomDatesGenerator}
6
7 val sci = new SciSparkContext(sc)
8
9 val dSet = sci.netcdfDataset("/path/to/directory/2014.10.box.atls.nc", List("time", "longitude",
10 "latitude", "sf"), 1)
11 val yyy = dSet.selDay("21,22,23")
12
13 yyy.plot3D("sf")
14
15
```

Abbildung 3.3. Beispielablauf einer Datenverarbeitung in *Zeppelin*.

¹ https://de.wikipedia.org/wiki/Uniform_Resource_Identifier

Die Ergebnisse dieser Datenanfrage werden in *dSet* - ein *CdoDataset*-Objekt mit Variablen *time*, *longitude*, *latitude* und *sf* zurückgegeben, welche zuvor in den entsprechenden Parametern bei der Anfrage eingegeben wurden. Nach der Erzeugung eines *CdoDataset*-Objektes mit aus der *NetCDF*-Datei abgelesenen Datensätzen, welche mit Hilfe einer Variablenliste beschränkt wurden, werden gewünschte Teilmenge von Datensätzen ausgewählt. Diese Auswahl lässt sich mit Hilfe eine Verwendung der Selektierungsmethode *selDay* (Zeile 11) aus sämtlichen Daten nur die Datensätze auswählen, die in einer Zeitspanne der Tagen des Monats liegen, in diesem Fall sind das 21, 22 und 23 Tage jedes Monats. Eine Visualisierung der gebliebenen Datensätzen wird durch eine Anwendung von *plot3D*-Funktion mit eingegebenem Parameter ermöglicht. Der Parameter definiert die Variable mit dreidimensionalen Nutzdaten (Zeile 13), die bei der Erzeugung von einem Grafikdiagramm verwendet werden.

Die Abbildung 3.4 präsentiert eine grafische Darstellung von Datenergebnisse, welche aus dem Ablauf der Datenverarbeitung aus der Abbildung 3.3 herausgekommen sind. Die Abbildung 3.4 zeigt eine dreidimensionale Oberflächendiagramm mit entsprechenden Axis-Bezeichnungen. Der Titel der Grafik wird aus dem Attributen der *NetCDF*-Datei zusammengebaut und deutet im gegebenen Fall die Menge von Niederschlägen wie Schnee. Rechts vom Oberflächendiagramm wird eine Skala der Farben dargestellt, die für bestimmte Menge von Niederschlägen benutzt werden. Unten lässt sich ein Datum mit Hilfe von Knöpfen aus gegebener Reihe von Tagen des Monats auswählen. Das Grafikdiagramm kann auch als eine Bild-Datei in dem *JPEG*-Format exportiert werden.

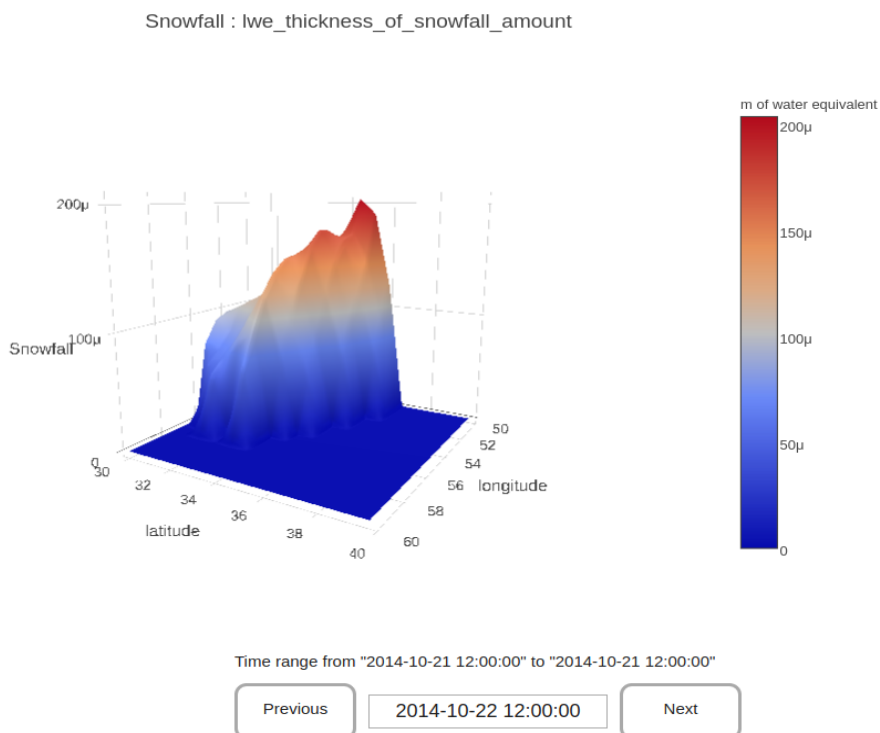


Abbildung 3.4. Grafikdiagramm dreidimensionaler Variable.

Zusammenfassung

In diesem Kapitel wurden technisch angeforderte Komponenten des entwickelten Systems sowie ein grundlegendes Modell der Zusammenarbeit zwischen denen vorgestellt. Dabei wurden Datenverarbeitungsprozesse erläutert. Eine Datenanalyse und eine Visualisierung der Ergebnisse wurden auf einem Beispielablauf veranschaulicht.

4 Implementation

Dieses Kapitel beschreibt wichtige Konzepte der Implementation des entworfenen System und detailliert erläutert kurz im dritten Kapitel eingeführte Komponenten des Systemmodells. Zunächst wird in Abschnitt 4.1 eine ausführliche Vorstellung der entwickelten Funktionalität von CDO-Merkmalen gegeben. In Abschnitt 4.2 wird eine Implementierung des Stencil-Verfahrens erläutert. Eine Implementation der Frontend-Datenvisualisierung wird in Abschnitt 4.3 vorgestellt. Schließlich wird in Abschnitt 4.4 eine Erläuterung zu der parallelen Datenverarbeitung gegeben.

Die Anwendung wurde unter dem Betriebssystem *Linux Ubuntu 16.10* entwickelt. Die entwickelten Merkmale sind sowohl für das *Frontend* als auch für das *Backend* in der integrierten *Eclipse*-Umgebung entworfen und implementiert worden. Die ganze Programmierung für *SciSpark*-Komponente ist in *Scala* geschrieben. *HTML*, *JavaScript* und *CSS* wurden für die Implementierung der Datenvisualisierung des *Frontends* verwendet.

4.1 CDO-Merkmale

Ein wesentlicher Teil dieser Arbeit wird der Datenverarbeitungen auf der Basis von *CDO*-Merkmalen gewidmet. Dabei werden in dem Abschnitt 3.2.4 erwähnte *CDO*-Funktionen in *Scala*-Programmiersprache für die *SciSpark*-Datenstrukturen entwickelt. Im Folgenden werden wichtige Konzepte dieser Entwicklung vorgestellt.

Für die Realisierung von *Informationsoperatoren* wurden einige Hilfsmethoden Hilfsklassen geschrieben. Als erstes Ziel der Implementierung von *CDO*-Methoden wird eine Anpassung von den zeitlichen Dimensionen einer *NetCDF*-Datei zu dem *Unixzeit*¹-Format gesetzt, denn in meisten Fällen die zeitliche Dimensionen von *NetCDF*-Dateien mittels der Attributen entsprechender *NetCDF*-Variable auf unterschiedlicher Art beschrieben werden. Dieses Zeitformat ist ähnlich aber nicht immer gleich zu dem *Unixzeit*-Format. Beispielsweise können zeitliche Dimensionen mit vergangenen Minuten seit dem ersten Oktober 1900, 00:00 Uhr Berliner Zeit sich bezeichnen lassen. Deswegen wird eine Ausgleichsmethode

¹ *Unixzeit* - die vergangenen Sekunden seit Donnerstag, dem 1. Januar 1970, 00:00 Uhr UTC
[<https://de.wikipedia.org/wiki/Unixzeit>]

verwendet, die sämtliche Zeitformaten in das *Unixzeit*-Format umwandelt. Dadurch wird eine allgemeine Ausführung der verlangten Verarbeitungsprozessen, basierenden auf der Zeitdimension, ermöglicht. Dafür wird die Funktion *getUnixGMTTimeVar* implementiert, welche sich auf der Basis einer Analyse von Attributen der Zeitdimension die oben beschriebene Formatumwandlung ausführen lässt. Als ein Rückgabewert wird für diese Funktion die Hilfsstruktur *CdoTime* verwendet, welche folgende Information in eigenen Feldern beinhaltet:

- *unixStartPoint* – der Zahlwert beschreibt einen Ausgangszeitpunkt in einem *Unixzeit*-Format.
- *units* – die Einheiten der Zeit, in denen die Kalkulation auf der Zeitspanne geführt wird.
- *dataArray* – die Elemente gegebener Zeitspanne vor der Umwandlung.
- *unixdataArray* – die entsprechenden Elemente gegebener Zeitspanne im *Unix*-Format.

Die beschriebene *getUnixGMTTimeVar*-Hilfsmethode wird in allen implementierten *CDO*-Merkmalen verwendet, die mit den Zeitdimensionen zusammenarbeitet.

Als nächste Herausforderung der Implementierung wird die Verarbeitung von Nutzdaten der *NetCDF*-Variablen beschrieben, die in einem *CdoDataset* als eine *HashMap*-Indexstruktur¹ von *Variable*-Objekten umfasst werden. Aus diesem *Variable*-Objekt, das in Abschnitt 3.2.2 vorgestellt wurde, ergibt sich die Speicherung von Nutzdaten der *NetCDF*-Variable in einem *AbstractTensor* und eine Liste in dieser Variable vorhandener Dimensionen. Von der technischen Seite werden die Nutzdaten in einem *AbstractTensor* als ein flaches (eindimensionales) Array von *Double*-Werten gespeichert. Deswegen müssen verlangte Indexe des flachen Arrays auf der Basis von Dimensionen der *NetCDF*-Variable berechnet werden. Zum Beispiel werden bei der Implementierung von der *selLonLatBox*-Methode die Inhaltswerte vorhandener *Variablen* geprüft, ob die in dem Spektrum der Grenzwerten von Längengrad- und Breitengrad-Dimensionen liegen. Dabei werden Indexe der Dimensionelementen (Längengrad und Breitengrad) definiert, die die verlangte Spektrumsvoraussetzung erfüllen. Auf der Basis dieser Dimensionsindexten werden entsprechende Datensätze selektiert. Ähnliche Verarbeitungsmodelle werden für die Implementierung meisten *CDO*-Merkmalen aus dem Abschnitt 3.2.4 verwendet.

Alle in dem *CdoDataset* implementierten *CDO*-Methoden liefern ein neues *CdoDataset*-Objekt mit angemessen berechneten Werten und Dimensionen des Datensatzes zurück. Somit wird eine Verwendung der Kombination verschiedener *CDO*-Merkmalen ermöglicht.

¹ *Hashtabelle* ist eine spezielle Indexstruktur, um Datenelemente in einer großen Datenmenge aufzufinden [<https://de.wikipedia.org/wiki/Hashtabelle>].

4.2 Stencil Algorithmen

Bei der Implementierung des *Stencil*-Algorithmuses wurde in dem *SciSpark*-Framework das Verfahren der Abstraktion für die Beschreibung benutzerdefinierter Berechnungsfunktion verwendet. Dieses Verfahren regelt sämtliche Anforderungen, die bei der Implementation der Berechnungsfunktion berücksichtigt werden müssen. Dafür wird vom Benutzer verlangt, eine Abstraktionsschnittstelle eines benötigten dimensional Datenmodells zu erweitern, und innerhalb dieser Erweiterung die vom Benutzer angeforderte Berechnungsoperationen zu implementieren. Einer Aufruf der benutzerdefinierter Funktion wird mit Hilfe der Abstraktionsschnittstelle innerhalb einer der *CdoDataset*-Merkmalen realisiert: *IStencil1DPattern*, *IStencil2DPattern* und *IStencil3DPattern*. Für einen einfacheren Schleifenablauf über die Elemente angeforderter Variable werden die zwei- und dreidimensionalen Variablendaten in ein Array entsprechender Dimensionalität umgewandelt. In der Auflistung 4.1 wird einen Berechnungsprozess des *Stencil*-Verfahren für eine dreidimensionale Variable implementiert, wobei in der Zeile 2 ein Objekt *stencilObj* übergeben wird, welches die *IStencil3DPattern*-Abstraktion implementiert. Innerhalb der Schleifenabläufe (Zeilen 4 bis 10) werden neue Werte aktueller Elementen vom dreidimensionalen Array nach der Methode *applyFunction* (Zeile 7) berechnet, die vom Benutzer in dem *stencilObj*-Objekt definiert wird. Die Implementation sowie ein Verwendungsbeispiel *IStencil3DPattern*-Abstraktion werden in der Auflistung 4.2 detailliert beschrieben.

```
1 ...
2 def stencil3DPattern(stencilObj: IStencil3DPattern,
                      step: Int,
                      variableName: String): CdoDataset = {
3   ...
4   //Aktualisierung des 3D-Arrays
5   for(a<-step until (x - step) ){
6     for(b<-step until (y - step) ){
7       for(c<-step until (z - step) ){
8         newVar3D(a)(b)(c) = stencilObj.applyFunction(var3DArr, a, b, c, step)
9       }
10    }
11  }
12 }
13 ...
```

Auflistung 4.1. Aufruf der benutzerdefinierter Funktion im *Stencil*-Verfahren für eine dreidimensionale Variable.

Hier wird ein Beispiel der Anwendung des *Stencil*-Verfahrens auf dem dreidimensionalen Datenmodell der *NetCDF*-Variable präsentiert. In den Zeilen 2 bis 7 folgt eine Implementation der *IStencil3DPattern*-Abstraktionsschnittstelle, wobei die benutzerdefinierte Funktion *applyFunktion* (Zeile 4) beschrieben wird. Nach dem die *NetCDF*-Datei

in das *dSet*-Objekt erfolgreich abgelesen wurde (Zeilen 8 und 9), wird die *stencil3DPattern*-Methode des *Stencil*-Verfahrens in Zeile 11 ausgeführt.

```
1 ...
2 class Stencil3dImpl extends IStencil3DPattern{
3     def applyFunction(A: Array[Array[Array[Double]]], x:Int, y: Int, z: Int,
4                                     step: Int) :Double = {
5         val t = A(x)(y)(z) + 1.0
6         t
7     }
8 val dSet : CdoDataset = sc.cdoNetcdfDataset("/path/to/atls14-CyG11B-sf.nc",
9                                             List("time", "longitude", "latitude", "sf"), 1)
10 val step = 1 // Menge freigelassener Elementen auf den Grenzen des Arrays
11 val result = dSet.stencil3DPattern(new Stencil3dImpl(), step, "sf")
12 ...
```

Auflistung 4.2. Anwendung des *Stencil*-Algorithmuses.

Die Implementationen des *Stencil*-Verfahrens für ein- und zweidimensionale *NetCDF*-Variablen werden nach gleichen Mustern realisiert.

4.3 Datenvisualisierung

Das entwickelte System realisiert die Visualisierung von dreidimensionalen *NetCDF*-Variablen mit Hilfe des *plotly.js*-Frameworks. Dafür werden entsprechende Daten innerhalb des *CdoDataset*-Objektes aus *Scala*-Programmiersprache in *JavaScript* konvertiert. Dieser Prozess wird von der *plot3D*-Funktion übernommen. Dabei werden *Variablen*-Objekte in mehrdimensionale *JavaScript* Arrays umgewandelt. Die Übermittlung der Ergebnisse von *SciSpark*-Framework zu der *Fronten*-Oberfläche des *Apache Zeppelin* wurde durch textuelle Darstellung von komplexen Objekten ermöglicht.

Wie es schon in dem zweiten Kapitel dieser Arbeit erwähnt wurde, beinhaltet das *Zeppelin*-System eine Liste einiger vorintegrierten Interpretermodulen. Zu dieser Liste gehört auch der *HTML*-Interpreter. Das *HTML*-Interpretermodul ermöglicht vollständige Ausführung sämtlicher *HTML*-Codes und verschiedener Webskripten. Durch eine Verwendung der *HTML*-Direktive „%html“ lässt *Apache Zeppelin* alle nachstehende Ergebnisausgänge als ein *HTML*-Code interpretieren. Diese Gegebenheit wird ausgenutzt um eine webbasierte interaktive Datenvisualisierung zu ermöglichen. Dafür werden einigen *HTML*-Elementen erstellt und beim Präsentieren von Ergebnissen einer Datenverarbeitung in dem Ausgangsfeld der *Zeppelin*-Oberfläche automatisch platziert.

In der Auflistung 4.3 wird eine Implementierung von *HTML*-Elementen dargestellt, die für eine Darstellung eines dreidimensionalen Oberflächendiagramm mit Hilfe von der *plotly.js*-Bibliothek benötigt werden. Zunächst wird in erster Zeile durch eine *HTML*-Direktive (Zeile

1 der Auflistung 4.3) des *HTML*-Interpreters initialisiert. Danach lässt sich ein *div*-Wrapper-Element (Zeile 2) mit einer Initialisierung der *plotly.js*-Bibliothek (Zeile 3) und einem *div*-Container für eine Diagrammdarstellung (Zeile 4) beschreiben.

```
1 %html
2 <div id="plot_wrap">
3   <script type="text/javascript" src="plotly-latest.min.js"/>
4   <div id="myDiv3D" style="width: 100%; height: 100%" />
5 </div>
```

Auflistung 4.3. *HTML*-Elemente für eine Datenvisualisierung im Frontend.

Die *plotly.js*-Bibliothek ermöglicht eine Erzeugung von einem Oberflächendiagramm mit der Methode *newPlot*. Die Auflistung 4.4 stellt einen Quellcode der Anwendung dieser Methode dar. Hier sieht man, dass die Syntax der *newPlot*-Funktion drei Parametern verlangt (Zeile 21). Mit erstem Parameter wird in die Funktion der Identifikator vom *div*-Element übergeben, in welchem das erzeugte Diagramm platziert wird. Mit dem zweiten Parameter werden sämtliche Nutzdaten an die Funktion übermittelt.

```
1 var layout = {
2   title : 'Snowfall : lwe_thickness_of_snowfall_amount',
3   autosize : true,
4   width : 900,
5   height : 600,
6   scene : {
7     xaxis : {
8       title : 'longitude',
9       autotick : true
10    },
11    yaxis : {
12      title : 'latitude',
13      autotick : true
14    },
15    zaxis : {
16      title : 'Snowfall',
17      autotick : true
18    }
19  }
20 };
21 Plotly.newPlot('myDiv3D', data_tensor, layout);
```

Auflistung 4.4. Erzeugung eines Grafikdiagramm.

Der dritte Parameter definiert eine Gestaltung des erzeugten Diagramms, womit sämtliche Grafikbezeichnungen wie Grafikachsen und Grafikmaßangaben beschrieben werden.

Für eine bequeme Umschaltung auf der Zeitspanne werden *Previous*- und *Next*-Knöpfen eingeführt. Die Funktionalität dieses Features lässt sich mit Hilfe von *jQuery*-Methoden implementieren, die auf dem *onClick*-Ereignis basieren und innerhalb dieser Beschreibung nicht vorgestellt werden.

4.4 Parallele Datenverarbeitung

Das SciSpark-Framework ermöglicht angeforderte Verarbeitungsprozesse sowohl seriell als auch parallel durchzuführen. Bei einer seriellen Datenverarbeitung lassen sich *NetCDF*-Dateien im Einzelnen in *SciSpark* mittels *CdoDataset*-Objekten bearbeiten. Die Auflistung 4.2 (Seite 54) präsentiert einen seriellen Ablauf des *Stencil*-Verfahrens für die *atls14-CyG11B-sf.nc* Datei. Für eine Parallelisierung dieses Ablaufes werden einige Anpassungen im Code gemacht. In der Auflistung 4.5 wird ein paralleles Verarbeitungsmodell des gleichen *Stencil*-Verfahrens gezeigt, wobei das Ablezen von *NetCDF*-Dateien mit Hilfe der *netcdfCdoDatasetList*-Funktion in ein *RDD*-Array von *CdoDataset*-Objekten durchgeführt wird (Zeilen 8 und 9). Der Datenverarbeitungsprozess vom *rddSets*-Array wird mit Hilfe der *map*-Funktion von *SciSpark* ausgeführt, welche sämtliche *CdoDataset*-Objekte innerhalb eines *RDD*s spaltet und eine parallele Verarbeitung mit Hilfe der implementierten in *CdoDataset* Merkmalen organisiert. Als ein Rückgabewert wird ein *RDD*-Array mit aktualisierten *CdoDatasets* zurückgeliefert.

```
1 ...
2 class Stencil3dImpl extends IStencil3DPattern{
3     def applyFunction(A: Array[Array[Array[Double]]], x:Int, y: Int, z: Int,
4                       step: Int) :Double = {
5         val t = A(x)(y)(z) + 1.0
6         t
7     }
8 }
9 val rddSets : RDD[CdoDataset] = sc.netcdfCdoDatasetList("/path/to/list_of_files.txt",
10 List("time", "longitude", "latitude", "sf"), 1)
11 val step = 1 // Menge freigelassener Elemente auf den Grenzen des Arrays
12 val resRddSets = rddSets.map{ dSet =>
13     dSet.stencil3DPattern(new Stencil3dImpl(), step, "sf")
14 }
15 ...
```

Auflistung 4.5. Paralleles Verarbeitungsmodell des *Stencil*-Verfahrens.

Auf der Basis dieses Verarbeitungsmodells können alle entwickelte Merkmale des *CdoDatasets* auch für die parallele Datenverarbeitung angewendet werden.

Zusammenfassung

In diesem Kapitel wurden Implementationen wichtiger Konzepte des entwickelten Systems beschrieben. Dabei wurden Datenverarbeitungsprozesse der implementierten CDO-Operatoren und der Stencil-Verfahren detailliert erläutert. Danach wurde eine Visualisierung der Daten in JavaScript- und HTML-Codes dargestellt. Schließlich wurde eine Erläuterung zu der parallelen Datenverarbeitung gegeben.

5 Evaluation

In diesem Kapitel werden Leistungscharakteristiken des entwickelten System ausgewertet. Zunächst wird in Abschnitt 5.1 die Testumgebung vorgestellt. Die Vorgehensmodelle der Experimentmessungen werden in Abschnitt 5.2 beschrieben. Abschnitt 5.3 präsentiert die Messergebnisse von einzelnen Experimenten. Zum Schluss werden Ergebnisse der durchgeführten Experimenten im Abschnitt 5.4 ausgewertet.

5.1 Testumgebung

Bei der Durchführung von Experimenten wurden das entwickelte System sowie Datenquellen auf dem Forschungscluster der Arbeitsgruppe Wissenschaftliches Rechnen der Universität Hamburg installiert. Die charakteristische Eigenschaften des Clusters sind:

| | |
|------------------|--|
| CPU: | AMD Opteron(tm) Processor 6344 @ 2,6 GHz |
| Arbeitsspeicher: | 257 GByte |
| Betriebssystem: | Ubuntu 16.04.1 LTS x86_64 |

Das entwickelte System wurde im *Standalone-Modus* mit drei *Worker*-Threads eingerichtet. Die *Worker*-Threads sind für die Berechnungsprozesse zuständig, somit werden die Berechnungsaufgaben des Systems über die drei Knoten verteilt. Die Benutzeroberfläche ist clientseitig und in jedem Webbrowser aufrufbar. Somit verwendet die Komponente den Rechner des Nutzers und alle seine Ressourcen. Für die Durchführung der Experimente werden *Chrome*-Webbrowser der Version 55.0.2883.87 (64-bit) für Linux-Betriebssysteme sowie die Kommandozeilenanwendung *CDO* der Version 1.7.2 benutzt.

5.2 Vorgehensmodell und Experimente

Das Ziel dieser Auswertung ist die Untersuchung der Ausführungszeiten der Datenverarbeitungsprozessen, die aus der Weboberfläche des entwickelten System angefordert sind. Die Ergebnisse werden nach der Berechnungsart (parallel/seriell) miteinander verglichen. Während dessen werden Abläufe der entwickelten *CDO*-Merkmale und des *Stencil*-Verfahrens begutachtet, die für die *NetCDF*-Daten ausgeführt werden. Jeder Testablauf wird

drei Mal wiederholt. Auf der Abbildung 5.1 wird einen Ablauf des Tests vorgestellt. Hierbei wird der Fokus auf der Frontend- und Berechnungskomponenten gelegt.

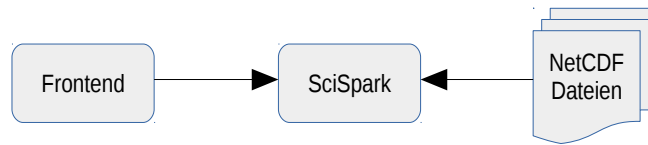


Abbildung 5.1. Testablauf mit grundlegenden Komponenten des Systems.

Auf der Basis von oben erwähnten Annahmen wird eine gesamte Ausführungszeit nach der Formel berechnet:

$$t_{\text{gesamt}} = t_{\text{frontend}} + t_{\text{SciSpark}} + t_{\text{NetCDF}} \quad (\text{Formel 5.1})$$

Wobei t_{frontend} eine Ausführungszeit in der Frontend-Komponente ist, t_{SciSpark} beschreibt die benötigte Zeit für eine Datenverarbeitung in SciSpark, t_{NetCDF} definiert die Zeit, die das Ablesen von *NetCDF*-Dateien aus einer Datenquelle benötigt. Im Umfang dieser Arbeit werden Experimente nur auf lokalen Datenquellen durchgeführt. Die Ausführungszeiten werden für die Datenübergaben über das Netzwerk für die Systemarchitektur mit Remotedatenquellen nicht erläutert.

Um die einzelnen Werte aus Formel 5.1 zu berechnen, werden im Folgenden unterschiedliche Tests durchgeführt. Dafür werden folgende Analyseprozesse ausgeführt:

- Datenverarbeitung mittels *SciSpark*-Funktionalität
- Datenvisualisierung mittels Webschnittstelle

Als Eingangsdaten werden für die Experimente folgende *NetCDF*-Dateien verwendet:

- *atls14-CyGl1B.nc* (1,8 GB, sieben Datensätze, dreidimensional 1096 x 241 x 480)
- *atls14-CyGl1B.1979.nc* (591,1 MB, sieben Datensatz, dreidimensional 365 x 241 x 480)
- *atls14-CyGl1B.1996.nc* (592,8 MB, sieben Datensatz, dreidimensional 366 x 241 x 480)
- *atls14-CyGl1B.2014.nc* (84,5 MB, sieben Datensatz, dreidimensional 365 x 241 x 480)
- *atls14-CyGl1B-sf.nc* (253,6 MB, ein Datensatz, dreidimensional 1096 x 241 x 480)
- *atls14-CyGl1B-sf.1979.nc* (84,5 MB, ein Datensatz, dreidimensional 365 x 241 x 480)
- *atls14-CyGl1B-sf.1996.nc* (84,7 MB, ein Datensatz, dreidimensional 366 x 241 x 480)
- *atls14-CyGl1B-sf.2014.nc* (84,5 MB, ein Datensatz, dreidimensional 365 x 241 x 480)

Die *atls14-CyG11B.nc*-Datei beinhaltet einen Satz von sieben Variablen. Die *atls14-CyG11B-sf.nc* fasst nur die Daten einer *sf*-Variable der *atls14-CyG11B.nc*-Datei. Mit diesem Größenunterschied wird eine Differenz in den Verarbeitungsabläufen erschafft. Um eine Abweichung zwischen den Ausführungszeiten der parallelen (in *RDD*) und seriellen Datenverarbeitungen festzustellen, werden *atls14-CyG11B-sf.1979.nc*, *atls14-CyG11B-sf.1996.nc* und *atls14-CyG11B-sf.2014.nc* sowie *atls14-CyG11B.1979.nc*, *atls14-CyG11B.1996.nc* und *atls14-CyG11B.2014.nc* verwendet, die Datensätze von welchen im entsprechenden Zusammenhang mit den Datensätzen der *atls14-CyG11B-sf.nc* und *atls14-CyG11B.nc* Dateien übereinstimmen. Damit wird eine Verarbeitung dieser Dateien innerhalb eines *RDD*-Arrays ermöglicht.

Um Ausführungszeiten der Datenverarbeitung analysieren zu können werden einige Benchmarkszenarios erschaffen, in denen das entwickelte System mit den Verarbeitungsprozessen belastet wird:

- *Szenario 1 (CDO)*.

Als ein Benchmarkszenario wird für die Auswertung der *CDO*-Merkmale ein Beispielscode der *Pipeline*-Ausführung aus Auflistung 2.6 (Seite 30) mit kleiner Anpassung genommen, sodass sich die Dateien für die Experimente nach Variablenmenge unterscheiden. Die Anpassung entsteht aus einem Verzicht vom *selvar*-Operator. Dabei wird die Syntax der *CDO*-Befehlen mit Hilfe der implementierten in *SciSpark* *CDO*-Merkmale beschrieben. Die *Pipeline*-Ausführungen werden sowohl im seriellen als auch im parallelen Modus getestet. Die Auflistung 5.1 stellt einen Beispielscode des Tests für serielle Datenverarbeitung dar. Hier wird die *Pipeline*-Ausführung in der Zeile 4 implementiert.

```

1 ... // Test für serielle Verarbeitung
2 val dSet : CdoDataset = sc.cdoNetcdfDataset("/path/to/netcdf_file.nc",
3                                     List("time", "longitude", "latitude", "sf"), 1)
4 val resSerial = dSet.monMean().merAvg().zonAvg()
5 ...

```

Auflistung 5.1. Code des seriellen Testablaufes.

In der Auflistung 5.2 wird ein Beispielscode des Tests für parallele Datenverarbeitung vorgestellt. Dabei werden sämtliche Datensätze in ein *RDD*-Array abgelesen (Zeilen 3 und 4). Die *Pipeline*-Ausführung wird mittels *map*-Funktion für jedes vorhandenen *CdoDataset*-Objekt im *RDD*-Array realisiert (Zeilen 5 bis 7).

```

1 ...
2 // Test für parallele Verarbeitung innerhalb RDD
3 val rddSets : RDD[CdoDataset] = sc.netcdfCdoDatasetList("/path/to/list_of_files.txt",
4                                     List("time", "longitude", "latitude", "sf"), 1)
5 val resRddSets = rddSets.map{ dSet =>
6     dSet.monMean().merAvg().zonAvg()
7 }
8 ...

```

Auflistung 5.2. Code des parallelen Testablaufes in *RDD*.

- *Szenario 2 (Stencil).*

Für die Auswertung des *Stencil*-Verfahren wird eine Filterfunktion gleitender Mittelwerte für dreidimensionalen Datensätzen implementiert. Dabei werden Ausführungszeiten der Verarbeitungen von Datensätzen innerhalb eines *RDD*-Arrays und ohne Verwendung der *RDD*-Datenstruktur miteinander verglichen. Die Testcodes basieren auf der Implementation der Auflistung 4.2 (Seite 54) für einen seriellen Durchlauf und der Auflistung 4.5 (Seite 56) für einen parallelen Durchlauf. Einzeln Unterschied ist die Implementation der benutzerdefinierter Funktion *applyFunction*. In aktuellen Experimenten trägt die Funktion einen anderen Berechnungslastfall.

Auf Grundlagen oben beschriebenen Szenarios werden Experimente durchgeführt, die im Folgenden detailliert erläutert werden.

5.2.1 Experiment 1: Verarbeitung der CDO-Szenarios

Im ersten Experiment wird eine Ausführungszeit gemessen, die für die Verarbeitung von den *NetCDF*-Dateien für *SciSpark* benötigt wird. Bei ersten Vermessungen werden *NetCDF*-Dateien *atls14-CyG11B.nc* und *atls14-CyG11B-sf.nc* als einzelne Datensätze verarbeitet. Auf der Abbildung 5.2 wird das Schema dieses Ablaufes der Datenverarbeitung angezeigt. Dabei werden größere *NetCDF*-Dateien im Einzelnen von *SciSpark* mittels *CdoDataset*-Objekten bearbeitet.

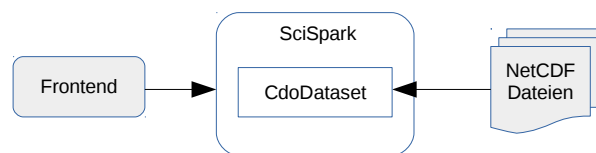


Abbildung 5.2. Einzelne Verarbeitung von *NetCDF*-Dateien

Im zweiten Modell der Datenverarbeitung wird besondere *RDD*-Datenstruktur von *Spark* verwendet. Damit werden Datenoperationen über die *Worker*-Knoten des Clusters verteilt.

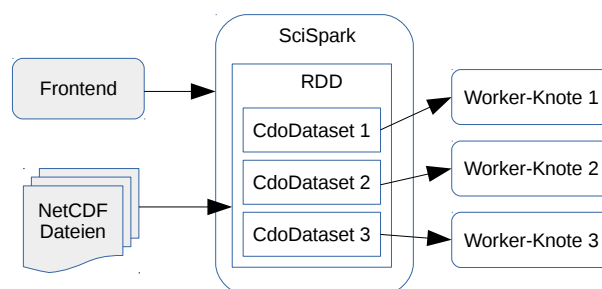


Abbildung 5.3. Verarbeitung von *NetCDF*-Dateien in *RDD*

Abbildung 5.3 zeigt, dass bei der Verwendung des *RDD*-Arrays die Dateien *tls14-CyGl1B-sf.a.nc*, *atls14-CyGl1B-sf.b.nc* und *atls14-CyGl1B-sf.a.nc* auf eigenen *Worker*-Knoten des *SciSpark*-Frameworks parallel bearbeitet werden. Beim zweiten Test lassen sich Dateien *atls14-CyGl1B.1979.nc*, *atls14-CyGl1B.1996.nc* und *atls14-CyGl1B.2014.nc* in einer *RDD*-Datenstruktur verarbeiten. Dabei liegen Erwartungen der Ausführungszeit von paralleler Datenverarbeitung in der Zeitspanne der längsten Ausführung der Verarbeitung in den *Worker*-Threads. In Abschnitt 5.3 wird es festgestellt, ob die Erwartungen verwirklicht wurden.

5.2.2 Experiment 2: Stencil Algorithmus

Bei der Durchführung vom zweiten Experiment werden *Stencil*-Manipulationen über die Dateien betrieben. Dabei werden Daten nach Verarbeitungsmodellen mit und ohne Verwendung der *RDD*-Strukturen, ähnlich zu Experiment 1, bearbeitet. Als benutzerdefinierte Berechnungsfunktion wird für das *Stencil*-Verfahren eine Filterfunktion gleitender Mittelwerte für dreidimensionalen Datensätzen genommen:

$$\dot{a}_{x,y,z} = \frac{a_{x-1,y,z} + a_{x,y-1,z} + a_{x,y,z-1} + a_{x,y,z} + a_{x+1,y,z} + a_{x,y+1,z} + a_{x,y,z+1}}{7} \quad (\text{Formel 5.2})$$

5.2.3 Experiment 3: Datenvisualisierung

Im dritten Experiment lassen sich verarbeitete Datensätze in der *Zeppelin*-Weboberfläche als ein dreidimensionales Oberflächendiagramm präsentieren. Die Erzeugung von Grafikdiagrammen wird mit Hilfe der *plotly.js-JavaScript*-Bibliothek ermöglicht. Dabei werden Daten aus *Scala*-Objekten in die *JavaScript*-Objekte konvertiert. In diesem Experiment werden Zeitaufwände gemessen, die für die Konvertierung, Datenübertragung und Datenvisualisierung benötigt werden. Für die Zeitmessung der Verarbeitungsprozesse in der Weboberfläche wird die Entwicklungsumgebung des *Google Chrome*-Browsers verwendet.

5.3 Messergebnisse

In diesem Abschnitt werden Messergebnisse analysiert, die bei der Durchführung von Experimenten aus dem Abschnitt 5.2 bekommen wurden. Für eine bessere Anschaulichkeit werden die Werte gemessener Parameter grafisch dargestellt.

5.3.1 Experiment 1: Verarbeitung der CDO-Szenarios

Nach dem Ablauf vom ersten Szenario werden Messergebnisse der Ausführungszeiten in der Tabelle 5.1 vorgestellt. Die Tabelle 5.1 zeigt, dass in *SciSpark* die Verarbeitungszeiten

sowohl der seriellen als auch der parallelen Abläufe mit den unterschiedlichen Gesamtgrößen der Dateien sehr ähnlich sind. Im Gegensatz zu diesen Ergebnissen sind die Ausführungszeiten der Datenverarbeitung zwischen parallelen und seriellen Abläufen in *SciSpark* gravierend unterschiedlich. Dabei verdoppelt sich die Ausführungszeit bei den seriellen Abläufen im Vergleich zu der Verwendung der *RDD*-Arrays, obwohl einen dreifachen Zeitgewinn für die *RDD*-Ausführung erwartet wurde. Dieses Ereignis lässt sich durch die Parallelisierung der Verarbeitungsprozessen erklären, wobei eine zuzügliche Zeit für die organisatorische Zwecke benötigt wird. Zeitunterschiede zwischen den Ausführungen der Szenarios in einem seriellen *SciSpark*-Prozess und in der *CDO*-Kommandozeilenanwendung als eine *Pipeline* liegen in Intervallen mehrerer Sekunden, jedoch mit der Verwendung der *Pipeline-Ausführung* gewinnt die Kommandozeilenanwendung bei dem *SciSpark*-Framework einige Sekunden an Verarbeitungszeit. Damit lässt sich das *RDD*-Verarbeitungsmodell nicht als das schnellste aus den beschriebenen erklären, jedoch wird in einer akzeptablen Zeit ausgeführt.

| Datensätze | Inputgröße | Ausführungszeit in <i>SciSpark</i> | Ausführungszeit in <i>CDO</i> | Outputgröße |
|---|----------------------------------|------------------------------------|-------------------------------|-------------|
| atls14-CyGl1B.nc | 1,8 GB | 18,9 s | 6,2 s | 4,3 kB |
| RDD von atls14-CyGl1B.1979.nc atls14-CyGl1B.1996.nc atls14-CyGl1B.2014.nc | 591,1 MB 592,8 MB 591,1 MB | 10,3 s | - | 9,5 kB |
| atls14-CyGl1B-sf.nc | 253,6 MB | 14,2 s | 1,1 s | 2,0 kB |
| RDD von atls14-CyGl1B-sf.1979.nc atls14-CyGl1B-sf.1996.nc atls14-CyGl1B-sf.2014.nc | 84,5 MB 84,7 MB 84,5 MB | 7,2 s | - | 5,9 kB |

Tabelle 5.1. Ergebnisse des Experimentes 1.

5.3.2 Experiment 2: Stencil Algorithmus

Im zweiten Experiment wurden die *NetCDF*-Dateien mit den Abläufen vom Stencil Algorithmus verarbeitet. Die Messergebnisse der Ausführungszeiten des Experimentes werden in der Tabelle 5.2 dargestellt. Aus der Tabelle 5.2 kann man die gleiche Tendenz feststellen, dass die Verarbeitungszeiten der parallelen Abläufe im Vergleich zu seriellen deutlich geringer sind. Man sieht auch, dass Zeitunterschiede zwischen Verarbeitungen von Dateien mit unterschiedlichen Gesamtgrößen sehr gering sind. Dieses Phänomen kann damit erklärt werden, dass wegen der schnellen Datenverarbeitungen viel Zeit für das Auslesen von *NetCDF*-Dateien aus den Datenquellen und für das Einpacken der Daten in die

CdoDataset-Objekte genommen wird. Die Ausführung von Prozessen im Arbeitsspeicher hat sehr geringen Zeitaufwand.

| Datensätze | Größe | Ausführungszeit |
|--|----------------------------------|-----------------|
| atls14-CyG11B.nc | 1,8 GB | 22,0 s |
| RDD von atls14-CyG11B.1979.nc atls14-CyG11B.1996.nc atls14-CyG11B.2014.nc | 591,1 MB 592,8 MB 591,1 MB | 12,7 s |
| atls14-CyG11B-sf.nc | 253,6 MB | 21,6 s |
| RDD von atls14-CyG11B-sf.1979.nc atls14-CyG11B-sf.1996.nc atls14-CyG11B-sf.2014.nc | 84,5 MB 84,7 MB 84,5 MB | 11,9 s |

Tabelle 5.2. Ergebnisse des Experimentes 2.

5.3.3 Experiment 3: Datenvisualisierung

Im dritten Experiment wurden Zeitmessungen der Verarbeitungsprozessen in der *Zeppelin*-Weboberfläche neben den Datenkonvertierungen innerhalb des *SciSpark*-Frameworks untersucht. Die Messergebnisse sind in der Tabelle 5.3 präsentiert.

| | | | |
|-------------------------------------|--------|---------|----------|
| Gesamtgröße der Datenergebnisse | 4,2 MB | 45,1 MB | 121,2 MB |
| Dauer der Konvertierung | 1,2 s | 12,5 s | 37,5 s |
| Dauer der Datenübertragung | 0,8 s | 1,2 s | 3,6 s |
| Größe des konvertierten Datensatzes | 4,3 MB | 45,2 MB | 121,3 MB |
| Dauer der Datenvisualisierung | 0,2 s | 1,3 s | 5,1 s |

Tabelle 5.3. Ergebnisse des Experimentes 3.

Aus der Ergebnissen ist eine lineare Abhängigkeit zwischen der Größe der Datenergebnisse und den Verarbeitungszeiten der Konvertierung, Datenübertragung und Datenvisualisierung zu sehen. Damit ergibt sich die Folgerung, dass die Verarbeitungszeiten der Visualisierungen dreidimensionaler Oberflächendiagramm von Ergebnissen mit den Datensätzen größer als 45 MB für die aktuellen Webbrowser enorm groß und ressourcenaufwendig sind. Die konsequente Schlussfolgerung zu diesen Ergebnissen und die Ideen zu der Problemlösung werden in Abschnitt 6.2 präsentiert.

5.3.4 Analyse der Verarbeitungszeiten

Um die gemeinsame Analyse der Messergebnisse von den Verarbeitungsprozessen auf der *NetCDF*-Dateien durchzuführen, werden die gemessenen Ausführungszeiten verschiedener

Umgebungen nach der ersten und zweiten Experimenten miteinander verglichen. Die Abbildung 5.4 stellt ein Säulendiagramm dieses Vergleichs.

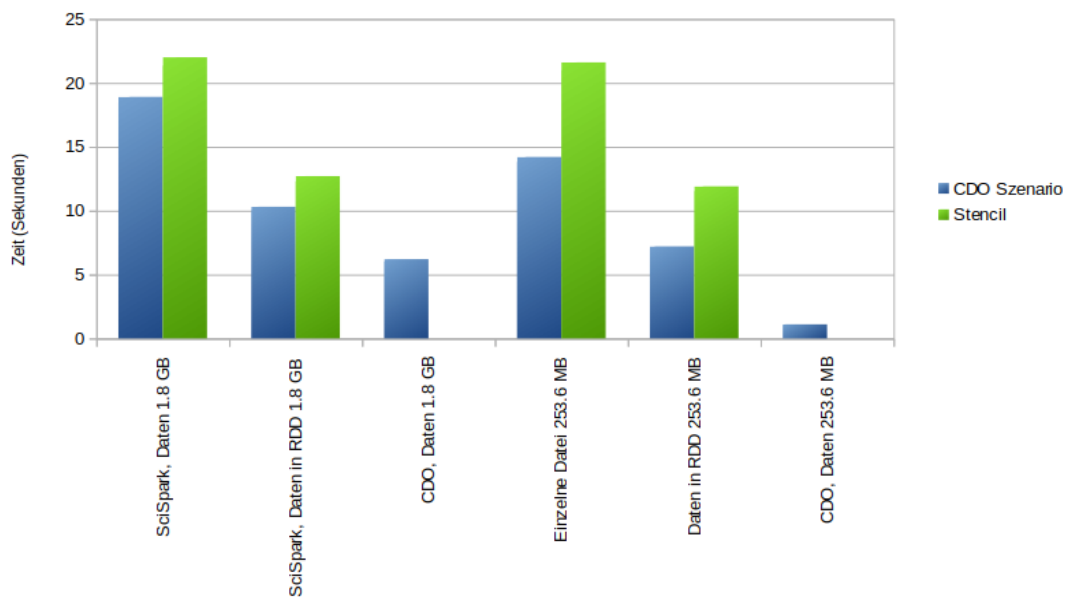


Abbildung 5.4. Ausführungszeiten der Datenverarbeitung mit verschiedenen Werkzeugen.

Im Diagramm kann man sehen, dass der Zuwachs der Verarbeitungseffizienz bei der Datenverarbeitung in *SciSpark* mit der Verwendung des *RDD*-Arrays im Vergleich zu der seriellen Prozessen auf allen getesteten Dateigrößen zwischen zweiundvierzig und fünfundfünfzig Prozent liegt. Zusätzlich zu den positiven Auswirkungen stellt man auch die negativen Seiten der Datenverarbeitung von *CDO*-Szenarios in *SciSpark* fest, wobei die Ausführungszeiten von *SciSpark* im Vergleich zur *CDO*-Kommandozeilenanwendung größer sind. Damit bleibt die *CDO*-Kommandozeilenanwendung mit der Verwendung der Pipeline das schnellste Werkzeug für die *NetCDF*-Dateien. Nichtsdestotrotz liefert das *SciSpark*-Framework akzeptable Ausführungszeiten der implementierten *CDO*-Merkmalen.

5.4 Fazit

Die Messergebnisse und Analysen aus Abschnitt 5.3 präsentieren folgende Feststellungen im Bezug auf die Verwendung des entwickelten System für die Datenverarbeitung von *NetCDF*-Dateien.

Beim Testen der Visualisierungsprozessen der Datenergebnisse in der *Frontend*-Komponente hat sich ergeben, dass die Weboberfläche die Datensätze bedeutsamer Größen verarbeiten können. Jedoch muss man drauf achten, dass das *Frontend* mit den möglichst kleinen Datensätzen arbeitet, denn sogar die Datenmengen im zweistelligen Megabyte-Bereich zu Problemen auch in den aktuellen Webbrowsern führen können.

Die entwickelte Funktionalität innerhalb der *SciSpark*-Komponente bietet hervorragende Leistung durch die Verteilung der angeforderten Datenverarbeitungen über die Cluster-Knoten. Der Leistungszuwachs ermöglicht bei der Verwendung von *RDD*-Strukturen eine Verringerung von den Verarbeitungszeiten bis zu fünfundvierzig Prozent. Die Ergebnisse dieser Arbeit lassen sich folgenden Herausforderungen der Verwendung des entwickelten System erklären:

- Implementation des *Stencil*-Verfahrens mit benutzerdefinierter Berechnungsfunktion für die *NetCDF*-Dateien.
- Akzeptable Ausführungszeit implementierten *CDO*-Merkmalen.
- Interaktive Visualisierung von Berechnungsergebnisse.
- Bequeme Weboberfläche mit der Funktion der Speicherung von Verarbeitungsabläufen.

Das in dieser Arbeit entwickelte System bietet eine leistungsfähige Plattform für die Analyse wissenschaftlicher *NetCDF*-Dateien mittels implementierten *CDO*-Funktionalität, *Stencil*-Algorithmus und Weboberfläche. Daneben lässt sich dieses Plattform leicht erweitern.

6 Zusammenfassung und Ausblick

In diesem Kapitel werden schließlich die Ergebnisse dieser Arbeit präsentiert. Abschnitt 6.1 bereitet einen kurzen Überblick der Arbeit. Zum Schluss wird in Abschnitt 6.2 ein Ausblick auf mögliche Erweiterungen des entwickelten Systems vorgestellt.

6.1 Zusammenfassung

Nachdem im ersten Kapitel einen Überblick auf die Tendenz der Entwicklung der Dateiformate für eine Aufbewahrung und Verarbeitung wissenschaftlicher Daten großer Mengen sowie die Basiskonzepte und Werkzeugen aktueller Datenverarbeitung als Motivation und schließlich die Ziele dieser Arbeit geschaffen wurden, erläuterte das zweite Kapitel die eingesetzten Technologien sowie verwandte Verarbeitungsanwendungen. Im dritten Kapitel wurde das Design des entwickelten Systems beschrieben. Dabei wurden grundlegende Architektur des Systems sowie verwendeten Datenstrukturen und vorkommende innerhalb des Systems Prozessabläufe vorgestellt. Kapitel 4 präsentierte Implementierungsmerkmale der wissenschaftlichen Datenverarbeitungsprozessen. Im fünften Kapitel wurde untersucht in wieweit das entwickelte System die gestellte Anforderungen erfüllt. Somit wurde auch die Nutzbarkeit von verwendeten Komponenten begutachtet, wobei es festgestellt wurde dass die Zusammenarbeit des *SciSpark*-Frameworkes mit der Weboberfläche von *Apache Zeppelin* ein leistungsfähiges und bequemes Datenverarbeitungssystem liefert.

Neben der Ermöglichung der Datenverarbeitung von *NetCDF*-Dateien innerhalb eines Webbrowsers, lässt das zusammengestellte System die wissenschaftlichen Verarbeitungsläufe als *Notebooks* für die Bewahrung und spätere Ausführung zu speichern. Außerdem lassen sich die Ergebnisse der Datenverarbeitungen als dreidimensionale Oberflächen-diagramme in der *Zeppelin*-Webschnittstelle darzustellen.

6.2 Ausblick

Im Laufe der Entwicklung des vorgestellten Systems wurden mehrere Ideen für eine weitere Erweiterungen der Funktionalität gesammelt. Für die Bearbeitungszeit wurde das Modell

mit Verwendung von *RDDs* realisierbarer Umfang definiert. Als mögliche Funktionalitätserweiterungen werden folgende Punkte angeboten:

- Implementierung weiterer *CDO*-Merkmalen aus Klassifizierung der Statistik, Interpolationen, Transformation, ...
- Integration verschiedener Algorithmen aus dem Maschinellen Lernen.
- Integration der Echtzeitdatenverarbeitung auf der Basis von *Spark Streaming*.
- Erstellung des Modells der Datenerarbeitung von einzelner *NetCDF*-Datei auf die *RDD*-Ebene. Damit wird eine Partitionierung vorhandener *NetCDF*-Variablen in die *RDD*-Struktur innerhalb eines *Scala*-Objektes gemeint, damit die Datenverarbeitungen auf der *RDD*-Basis parallel geführt werden kann.
- Übertragung der Datenvisualisierung in das *Backend* und eine Ermöglichung der Speicherung von den Visualisierungen in einem dreidimensionalen Dateiformat kleinerer Gesamtgröße sowie eine Darstellung dieser Visualisierungen in dem *Frontend*.

Sobald die oben erwähnten Punkten erfüllt werden, erhält das System eine breitere Palette an Werkzeugen der Datenanalyse und erhöht Effizienz der Verarbeitungsprozessen.

Literaturverzeichnis

- [UCAR] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, Ed Hartnett, Dennis Heimbigner, Ward Fisher. NetCDF Introduction and Overview, 2016 <http://www.unidata.ucar.edu/software/netcdf/docs/index.html>
- [GeoCons] Ben Domenico. NetCDF Binary Encoding Extension Standard: NetCDF Classic and 64-bit Offset Format. Open Geospatial Consortium 2011. <http://www.opengis.net/doc/IS/netcdf-binary/1.0>
- [ASFSpark] Apache Software Foundation. Spark Tutorial. 2016. <http://spark.apache.org/docs/latest/>
- [SparkStream] Spark Streaming Programming Guide. 2016. <http://spark.apache.org/docs/latest/streaming-programming-guide.html>
- [StencilWiki] Stencil code. 2017. https://en.wikipedia.org/wiki/Stencil_code#cite_note-Sloot-2
- [VonNeumann] Von Neumann neighborhood. 2017 https://en.wikipedia.org/wiki/Von_Neumann_neighborhood
- [AMPLab] Atchley Kattt. Spark-Lightning-Fast Cluster Computing. AMPLab der University of California in Berkeley 2011. <https://amplab.cs.berkeley.edu/projects/spark-lightning-fast-cluster-computing/>
- [RDDBerk] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. <https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>
- [ClCompBerk] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. Spark: Cluster Computing with Working Sets. University of California, Berkeley. 2011. <https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf>
- [LamArc] Shapira, Gwen. Building Lambda Architectur with Spark Streaming. 2014. <http://blog.cloudera.com/blog/2014/08/building-lambda-architecture-with-spark-streaming/>
- [SciSpGit] SciSpark Introduction. 2016. <https://github.com/SciSpark/SciSpark>

- [SciSparkApp] Rahul Palamuttam, Renato Marroquín Mogrovejo, Chris Mattmann, Brian Wilson, Kim Whitehall, Rishi Verma, Lewis McGibbney, Paul Ramirez. SciSpark: Applying In-memory Distributed Computing to Weather Event Detection and Tracking. 2015.
<http://geo-bigdata.github.io/2015/papers/S08216.pdf>
- [SkyND] Skymind. “ND4J: Scientific Computing for Java”. 2015.
<http://nd4j.org/about.html>
- [ScalaNLP] Hall, David. “Scala NLP: Scientific Computing, Machine Learning, and Natural Language Processing”. 2015.
<http://www.scalanlp.org/documentation/>
- [AZIntr] Apache Zeppelin Introduction. 2016.
<https://zeppelin.apache.org/docs/0.6.2/>
- [ZepVis] Using Zeppelin for Data Visualization at TubeMogul. 2016
<https://www.tubemogul.com/engineering/using-zeppelin-for-data-visualization-at-tubemogul/>
- [JN] The Jupyter Notebook. 2016.
<http://jupyter.org/>
- [HDFGr] HDF Group. Introduction to HDF5. 2006
<https://support.hdfgroup.org/HDF5/doc/H5.intro.html#Intro-FileOrg>
- [CDFnasa] NASA. General Information. 2016.
<http://cdf.gsfc.nasa.gov>
- [ASFHadoop] Apache Software Foundation. Welcome to Apache Hadoop! 2016
<http://hadoop.apache.org/>
- [CDO16] Uwe Schulzweida. CDO User’s Guide. 2016.
<https://code.zmaw.de/projects/cdo/embedded/cdo.pdf>
- [GDT] GDT netCDF conventions for climate data, version 1.3. 2016.
http://www-pcmdi.llnl.gov/drach/GDT_convention.html
- [COARDS] Conventions for the standardization of NetCDF files. 2016.
http://ferret.wrc.noaa.gov/noaa_coop/coop_cdf_profile.html
- [CF] CF Conventions and Metadata. 2016.
<http://cfconventions.org/>
- [OPeNDAP] OPeNDAP Software. 2016.
<https://www.opendap.org/index.php/software>
- [THREDDS] THREDDS Data Server (TDS). 2016.
<http://www.unidata.ucar.edu/software/thredds/current/tds/>

Abbildungsverzeichnis

| | |
|---|----|
| 2.1. Ablauf eines iterativen Algorithmus in Hadoop MapReduce..... | 16 |
| 2.2 <i>Apache Spark RDD-Datenstruktur</i> | 17 |
| 2.3 Ablauf eines iterativen Algorithmus in <i>Apache Spark</i> | 18 |
| 2.4 Nutzung von Datenquellen in <i>Standalone</i> -Modus von <i>Spark</i> | 20 |
| 2.5 Nutzung von Datenquellen in <i>Cluster</i> -Architektur von <i>Spark</i> | 21 |
| 2.6 Die <i>SciSpark</i> -Architekture..... | 22 |
| 2.7 Die <i>SciTensor</i> -Architekturen innerhalb des <i>sRDD</i> | 23 |
| 2.8 Die <i>SciDataset</i> -Architekturen innerhalb des <i>RDD</i> | 24 |
| 2.9 <i>Apache Zeppelin</i> Architekture..... | 26 |
| 2.8 Zweidimensionale <i>Von-Neumann-Nachbarschaft</i> mit Manhattan-Distanz von 1..... | 31 |
| 2.9 Dreidimensionale <i>Von-Neumann-Nachbarschaft</i> mit Manhattan-Distanz von 1..... | 32 |
| 3.1 Leistungsfähigkeit von <i>Spark</i> und <i>Hadoop MapReduce</i> | 35 |
| 3.2. Architektur des entwickelten Systems..... | 37 |
| 3.3 Beispielablauf einer Datenverarbeitung in <i>Zeppelin</i> | 48 |
| 3.4 Grafikdiagramm dreidimensionaler Variable..... | 49 |
| 5.1 Testablauf mit grundlegenden Komponenten des Systems..... | 58 |
| 5.2 Einzelne Verarbeitung von <i>NetCDF</i> -Dateien..... | 60 |
| 5.3 Verarbeitung von <i>NetCDF</i> -Dateien in <i>RDD</i> | 60 |
| 5.4 Ausführungszeiten der Datenverarbeitung mit verschiedenen Werkzeugen..... | 64 |

Listingübersicht

| | |
|---|----|
| 2.1 Ausgabe des <i>ncdump</i> -Kommando für eine <i>NetCDF</i> Datei..... | 11 |
| 2.2 Implementierung der Mapper-Schnittstelle für <i>Hadoop MapReduce</i> | 14 |
| 2.3 Implementierung der <i>Reducer</i> -Schnittstelle für <i>Hadoop MapReduce</i> | 15 |
| 2.4 Kalkulation der Wortfrequenz in <i>Hadoop MapReduce</i> | 15 |
| 2.5 Kalkulation der Wortfrequenz in <i>Spark</i> | 18 |
| 2.6 Auslesen von <i>NetCDF</i> -Daten in das <i>RDD</i> -Array von <i>SciDataset</i> -Objekten.. | 25 |
| 2.5 <i>CDO</i> -Einzelbefehle..... | 30 |
| 2.6 <i>CDO Pipeline</i> -Ausführung..... | 30 |
| 3.1 Datenstruktur von <i>CdoDataset</i> -Objekt..... | 42 |
| 3.2 Datenstruktur des <i>Variable</i> -Objektes..... | 42 |
| 3.3 <i>JSON</i> -Datenstruktur für die Datenvisualisierung im Frontend..... | 43 |
| 4.1 Aufruf der benutzerdefinierter Funktion im <i>Stencil</i> -Verfahren für eine dreidimensionale <i>Variable</i> | 53 |
| 4.2 Anwendung des <i>Stencil</i> -Algorithmuses..... | 54 |
| 4.3 <i>HTML</i> -Elemente für eine Datenvisualisierung im Frontend..... | 55 |
| 4.4 Erzeugung eines Grafikdiagramm..... | 55 |
| 4.5 Paralleles Verarbeitungsmodell des <i>Stencil</i> -Verfahrens..... | 56 |
| 5.1 Code des seriellen Testablaufes..... | 59 |
| 5.2 Code des parallelen Testablaufes in <i>RDD</i> | 59 |

Tabellenverzeichnis

| | |
|---|----|
| 3.1. Aspekte der Auswahl der Weboberfläche..... | 36 |
| 3.2. Aspekte der Auswahl der JavaScript-Bibliothek..... | 40 |
| 5.1 Ergebnisse des Experimentes 1..... | 62 |
| 5.2. Ergebnisse des Experimentes 2..... | 63 |
| 5.3. Ergebnisse des Experimentes 3..... | 63 |