



Vergleich dreier Implementationsvarianten für eine Analyse von Satellitenbildern

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Arts (B. A.)

HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT II
INSTITUT FÜR INFORMATIK

eingereicht von: Robin Ellerkmann
geboren am: 25.04.1992
in: Berlin

Gutachter: Prof. Johann-Christoph Freytag, Ph.D.
Dipl.-Inf. Mathias Peters

eingereicht am:

verteidigt am:

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	3
2.1	Grundlagen der Satellitenbildanalyse	3
2.1.1	Fernerkundung mithilfe des Landsat-Satellitensystems	3
2.1.2	Aufbereitung und Analyse von Satellitenbildern	4
2.2	Parallele Datenverarbeitungssysteme	5
2.2.1	Bedeutung und Eigenschaften von Big Data	6
2.2.2	Systeme zur massiv parallelen Datenverarbeitung	7
2.2.3	Apache Flink	10
2.2.4	Python	12
2.2.5	Vergleichsmetriken	13
3	Algorithmus zur Analyse von Pixelzeitreihen	15
3.1	Beschreibung des Algorithmus	15
3.2	Umsetzung des Algorithmus mit Apache Flink	20
3.2.1	Komplexitätsanalyse	21
3.2.2	Unterschiede zwischen der Nutzung der Java-Programmierschnittstelle und der Python-Programmierschnittstelle	21
3.3	Umsetzung des Algorithmus in Python	21
4	Evaluierung	23
4.1	Evaluierungskriterien	23
4.2	Versuchsbeschreibung	24
4.3	Auswertung	25
5	Fazit	26

Kapitel 1

Einleitung

In den vergangenen Jahren war ein massiver Zunahme des generierten Datenaufkommens zu beobachten [EMC14]. Viele Projekte, Unternehmen und Institutionen haben Zugriff auf eine gewaltige Menge an Daten. Diese wächst immer schneller an. 2004 analysierte Google circa 100 Terabyte pro Tag [DG04]. Bis zum Jahr 2008 war die täglich zu analysierende Datenmenge bereits auf 20 Petabyte angewachsen [DG08]. Das Sloan Digital Sky Survey, das ein Viertel des Himmels astronomisch erkundet, hat seit 1998 insgesamt 116 Terabyte an astronomischen Daten gesammelt [YAJEA⁺00, AAP⁺15]. Jede Nacht kommen circa 250 Gigabyte neu hinzu [Quelle: Herr Prof. Freytags VL, dort angegebene Quelle ist offline. Wie angeben?](#). Ein weiteres Beispiel ist das 1000 Genomes Project [Bak10], das zwischen 2008 und 2013 insgesamt 464 Terabyte Daten zum menschlichen Genom sammelte. Insgesamt werden die Datenmengen weiter stark zunehmen, für das Jahr 2020 wird eine weltweites Datenaufkommen von 44 Zettabyte prognostiziert [EMC14]. Diese Entwicklung offenbart diverse neue Herausforderungen bei der Speicherung, Verarbeitung und Analyse von Daten. Dabei spielt die möglichst schnelle Verarbeitung von stetig generierten Daten eine große Rolle. Diese muss im Gegensatz zur Verarbeitung bereits gespeicherter Daten abhängig vom aktuellen Datenaufkommen skalieren. Aktuelle Datenverarbeitungssysteme wie Apache Hadoop [Foue] und Apache Flink [Foud] bieten diese Möglichkeit der Datenflussanalyse und ermöglichen eine flexible Analyse der Daten. Kern dieser Systeme ist eine Implementierung des Map-Reduce Paradigmas [DG08] sowie die Nutzung von *User Defined Functions*. Diese ermöglichen eine parallele Abarbeitung von Arbeitsschritten in einem direkten, azyklischen Graphen. Der DAG wird zuvor aus dem vom User bereitgestellten Quellcode erzeugt. Die durch diese Architektur erreichbare, massiv parallelisierbare Ausführung der Datenanalyse ermöglicht die Nutzung von Clustern. Somit wird eine skalierbare Infrastruktur genutzt, die wiederum eine skalierte Nutzung der Datenverarbeitungssysteme ermöglicht. Diese Systeme können auch die weiterhin wichtige Nutzung und Analyse von bereits konsistent gespeicherten Daten unter Nutzung des parallelen Verarbeitungsansatzes durchführen. Dies ist insbesondere deshalb notwendig, da traditionelle Datenbanksysteme große Datenmengen nicht immer in akzeptabler Form und Verarbeitungszeit verarbeiten können [Jac09]. Das Hauptproblem bei der Verarbeitung von großen Datenmengen auf einzelnen Maschinen entsteht, wenn die zu verarbeitende Datenmenge die Hauptspeichergröße übersteigt. In diesem Fall müssen die nicht in den Hauptspeicher speicherbaren Daten zur Verarbeitungszeit nachgeladen werden, was die Verarbeitungszeit aufgrund der unterschiedlichen Beschaffenheit der verschiedenen Speicherebenen extrem verlängert. Um diese Speicherklappe zu umgehen, werden zunehmend parallelisierbare Ansätze der Datenverarbeitung verfolgt. Im Rahmen dieser Bachelorarbeit sollen demzufolge ein traditioneller und ein massiv parallelisierbarer Ansatz bei der Verarbeitung von großen Datenmengen untersucht werden. So soll eine Abschätzung der Leistungsfähigkeit, Vorteile und Nachteile beider Ansätze ermittelt werden. Der Vergleich beider Ansätze wird am Beispiel ei-

nes Algorithmus zur Approximierung von Pixelzeitreihen durchgeführt. Dieser wird im Rahmen des Projekts GeoMultiSens[GP] zur Analyse der Veränderung der Flora in einer geographischen Region genutzt. Dabei werden durch Landsat-Satelliten Satellitenaufnahmen bereitgestellt, die nach der Aufbereitung durch vorgestellte Algorithmen ausschnittsweise untersucht werden. Nach der Analyse werden anschließend mithilfe des Algorithmus auf Basis der approximierten Werte Prognosen zur weiteren Entwicklung der Flora der untersuchten Region gestellt. Dabei werden bei einer Analyse mehrere Szenenausschnitte derselben geographischen Region analysiert. Dabei müssen große Datenmengen verarbeitet werden, so dass sich die Nutzung eines aktuellen Datenverarbeitungssystems anbietet. Bei dieser Bachelorarbeit wird als Vertreter der massiv parallelisierbaren Datenverarbeitungssysteme Apache Flink genutzt.

Es werden drei unterschiedliche Implementierungen des Algorithmus untersucht, die sich hinsichtlich der eingesetzten Technologien und Programmiersprachen unterscheiden. Die zugrunde liegende Methodik, die der Algorithmus implementiert, ist bei allen untersuchten Varianten identisch. Als Basis wird die bereits implementierte und in der Praxis genutzte Python-Implementation genutzt. Sie sollte somit die untere Schranke der Leistungsmessungen darstellen. Die zweite und dritte Variante werden in Flink implementiert. Diese beiden Varianten unterscheiden sich bezüglich der genutzten Programmiersprache. [Zur Implementierung der zweiten Variante wird Java-Schnittstelle von Flink genutzt, zur Umsetzung von Variante drei die Python-Schnittstelle.](#) Schließlich werden alle drei Varianten unter identischen Bedingungen getestet. Dies bedeutet, dass sowohl die Testumgebung als auch die Testdaten identisch sein sollen. Dabei sollen alle Varianten sowohl auf einer leistungsfähigen Einzelmaschine als auch mit einem Cluster von Maschinen getestet werden. Ausgehend von den Untersuchungen und den ermittelten Ergebnissen soll nachfolgend eine Bewertung der drei Implementierungsvarianten des Algorithmus vorgenommen werden. Dabei sollen insbesondere die Größe der Ausgangsdatenmenge, die genutzte Hardware sowie die Größe der untersuchten Bildausschnitte in Bezug zu den Ergebnissen gesetzt werden.

Kapitel 2

Grundlagen

Die Basis für die Satellitenbildanalyse mittels Apache Flink bilden zum einen Konzepte aus der Geographie, zum anderen Strukturen und Vorgehensweisen aus der Informatik. Während die geographische Komponente insbesondere bei der Aufnahme der Satellitenbilder sowie bei der inhaltlichen Konzeption der Analysen sowie der Bereinigung der Daten vertreten ist, ist die Informatik für eine technisch korrekte und effiziente Umsetzung der geographischen Konzepte verantwortlich. Nachfolgend werden zuerst die geographischen Grundlagen der Fernerkundung erläutert. Dies umfasst insbesondere die technische Spezifikation der Aufnahmegeräte der eingesetzten Satelliten sowie wichtige Verfahren zur Datenaufbereitung sowie zur Datenanalyse. Anschließend wird ein Überblick über parallele Datenverarbeitungssysteme gegeben. Dieser umfasst auch die Eigenschaften von Big Data sowie eine konzeptuelle Beschreibung der Datenanalyseplattform Apache Flink. Abschließend wird Apache Flink aus der Entwicklerperspektive betrachtet. Dabei werden insbesondere Eigenschaften der Plattform beschrieben, die bei der Entwicklung von Programmen auf Basis von Flink von Bedeutung sind. Des weiteren wird die Programmiersprache Python betrachtet.

2.1 Grundlagen der Satellitenbildanalyse

2.1.1 Fernerkundung mithilfe des Landsat-Satellitensystems

Als Fernerkundung wird „die Gesamtheit der Verfahren zur Gewinnung von Informationen über die Erdoberfläche oder anderer nicht direkt zugänglicher Objekte durch Messung und Interpretation der von ihr ausgehenden (Energie-) Felder“ [Deu12] verstanden. Fernerkundungssatelliten verfügen über verschiedene Aufnahmesysteme, die durch multispektrale Messungen von emittierter elektromagnetischer Strahlung eine berührungsfreie Beobachtung der Erdoberfläche ermöglichen. Bei der multispektralen Messung werden von Sensoren registrierte spektrale Signaturen einzelnen Bereichen des elektromagnetischen Spektrums zugeordnet. Das Resultat sind mehrere spektrumsspezifische, simultan aufgenommene Satellitenbilder, die nur das aufgefangene Licht eines spezifischen Spektralbereichs, auch Spektralband genannt, zeigen. Die Art und Qualität der Aufnahmesensoren ist dabei abhängig vom Typ des Satelliten.

Die Ausgangsdaten für die Untersuchungen in dieser Bachelorarbeit wurden von Satelliten des Landsat-Satellitensystems aufgenommen. Der erste Landsat-Satellit Landsat 1 wurde 1972 gestartet. Seitdem wurden die Sensoren und die Satelliten kontinuierlich weiterentwickelt. Aktuell sind Landsat 7 und, im Rahmen der Landsat Data Continuity Mission, Landsat 8 im Einsatz. Landsat 8 nutzt in der aktuellen Generation zwei verschiedene Instrumente zur Fernerkundung. Den Operational Land Imager (OLI) und die Thermal Infrared Sensors (TIRS).

Der OLI erfasst emittierte elektromagnetische Strahlung im Spektralbereich von 0,433 μm bis 1,390 μm unterteilt in acht Spektralkanäle sowie einen panchromatischen Kanal. Es werden mehr als 7000 Detektoren pro Spektralband genutzt, um eine bessere Bildqualität zu bieten als frühere Systeme [MSWI04]. Neben den klassischen Farbspektren Blau, Grün und Rot nutzt Landsat-8 ein weiteres Band, das speziell für die Fernerkundung von Küsten genutzt wird. Außerdem verfügt Landsat 8 über drei Infrarotbänder, die nahes und mittleres Infrarotlicht registrieren, sowie ein weiteres Infrarotband, das auf die Beobachtung von Cirruswolken spezialisiert ist. Der panchromatische Kanal registriert elektromagnetische Strahlung mit Wellenlängen von 0,500 μm bis 0,680 μm . Dieser Spektralbereich entspricht etwa dem des menschlichen Auges. Aufgrund des, im Vergleich zu den einzelnen Farbfrequenzbändern, breiten abgedeckten Spektralbereichs ist eine höhere Auflösung der Bilder möglich.

Die Thermal Infrared Sensors (TIRS) [Cha11] umfassen zwei Thermalkanäle. Diese erfassen im Gegensatz zu den Multispektralkanälen elektromagnetische Emissionen mit Wellenlängen zwischen 10,30 μm und 12,50 μm , also langwellige Infrarotstrahlung. Dies ist insbesondere für die Beobachtung von Wolken nützlich. Die Kantenlänge der einzelnen Pixel beträgt 100 Meter. Diese kann nachträglich auf 30 Meter angeglichen werden, um eine bessere Kompatibilität mit den Aufnahmen der Multispektralbänder zu gewährleisten.

Um mithilfe der durch die Spektralbänder registrierten Werte die Vegetation eines geographischen Bereichs bestimmen zu können, wird ein Vegetationsindex berechnet. Ein bekannter Vegetationsindex ist der *Normalized Difference Vegetation Index*. Dieser bestimmt den Grad der Vegetation durch die Analyse der Werte des roten sichtbaren Spektralbereichs von 600-700 μm sowie des nahen Infrarotbereichs zwischen 700 und 1300 μm . Dabei wird ausgenutzt, dass Chlorophyll Licht im infrarotnahen Spektrum reflektiert. Durch die Messung dieses Bereichs lässt sich die Menge an Chlorophyll und somit ein dem entsprechender Grad an Vegetation ermitteln. Eine Schwäche des Normalized Difference Vegetation Index ist die fehlende Unterscheidbarkeit von wenig bewachsenen Flächen und Flächen mit krankhaften Pflanzen, die weniger Chlorophyll aufweisen. Aufgrund dieser Ungenauigkeit des NDVI werden zur Identifizierung von Waldflächen nach [ZWO12] zusätzlich zu den Daten der Spektralbänder 3, 4 und 7 auch die Daten der Bänder 1, 2 und 5 genutzt. Die Kombination der Informationen dieser Bänder ermöglicht eine genauere Bestimmung von Waldflächen durch die Berücksichtigung von Oberflächenreflektionen. Dies ist notwendig, um saisonale Unterschiede in die Analyse mit einzubeziehen [ZWO12, MHW⁺08].

Landsat 8 sendet pro Tag 400 Aufnahmen der Erdoberfläche, auch Szenen genannt, an die Bodenstation. Eine Aufnahme zeigt dabei eine geographische Region der Erde mit einer Ost-West-Ausdehnung von 185 Kilometer. Dies entspricht 100 nautischen Meilen. Die Nord-Süd-Ausdehnung einer Szene beträgt circa 174 Kilometer. Benachbarte Szenen überlappen sich dabei, so dass einige geographische Bereiche auf mehreren Szenen zu finden sind.

Durchschnittlich wird jede Region der Erde alle 16 Tage von Landsat 8 überflogen, so dass jährlich mindestens circa 22 Aufnahmen eines geographischen Bereichs gemacht werden [IDB12]. Hinzu kommen die von Landsat 7 angefertigten Aufnahmen.

Die von Landsat-Satelliten aufgezeichneten und übermittelten Bilder müssen zwecks diverser Korrekturen und Normalisierungen vor der Durchführung von Analysen aufbereitet werden.

2.1.2 Aufbereitung und Analyse von Satellitenbildern

Die durch die Landsat-Satelliten aufgezeichneten und an die Bodenstationen übermittelten Szenen müssen vor ihrer Nutzung aufbereitet werden. Dadurch wird im Allgemeinen die Bildqualität verbessert, da externe Störfaktoren und eventuelle interne Fehlfunktionen ausgeglichen werden können. Es wird dabei zwischen geometrischen und radiometrischen Aufbereitungen unterschieden.

Im Rahmen der geometrischen Aufbereitung sollen die Folgen einer eventuellen Fehlpositionierung des Satelliten korrigiert werden. Damit die Szenen sinnvoll analysiert werden können,

müssen sie korrekt und genau positioniert sein. Dies gilt insbesondere bei der Analyse einer Serie von Szenen derselben geographischen Region. Um eine normierte Positionierung einer Szene zu schaffen, werden die Satellitenbilder geokodiert. Dies bedeutet, dass jedem Pixel einer Szene eine entsprechende Koordinate eines geographischen Koordinatensystems zugewiesen wird. Dies kann beispielsweise durch die Anwendung der Paßpunkt-Methode oder eines Resampling-Verfahrens erreicht werden. Nur durch diese Normierung kann garantiert werden, dass positionsbezogene Daten aus verschiedenen Quellen zuverlässig den entsprechenden realen Positionen zugeordnet werden können. Zusätzlich zur notwendigen Geokodierung einer Szene sind möglicherweise weitere geometrische Korrekturen notwendig, um beispielsweise Verzerrungen zu entfernen.

Nachdem die Satellitenbilder geometrisch aufbereitet wurden, können sie bei Bedarf zusätzlich radiometrisch verbessert werden. Die Art der Verbesserungen ist dabei insbesondere von der geplanten Analyse abhängig und sorgt im Allgemeinen dafür, dass die der Analyse zugrundeliegenden Werte besser sichtbar gemacht werden. Zu den radiometrischen Verbesserungen gehören zum Beispiel atmosphärische Korrekturen wie das Entfernen von Wolken und Wolkenschatten oder von durch die Atmosphäre verursachten Verschlechterungen, die aus Interferenzen innerhalb der Atmosphäre zwischen Erdoberfläche und dem Satelliten resultieren. Techniken um diese Verbesserung zu erreichen sind beispielsweise das Strahlungstransfermodell, die bildbasierte atmosphärische Korrektur und die Histogramm-Minimum-Methode. Es ist individuell von der Szene und den zur Verfügung stehenden Metadaten abhängig, mit welcher Methode die nützlichste Verbesserung erreicht werden kann. Eine weitere radiometrische Aufbereitung ist die Kontraststreckung, die den Kontrast zwischen verschiedenen Farbwerten, die innerhalb eines Spektralbereichs auftreten, verbessert, um etwaige Unterschiede eindeutiger feststellen zu können [Pad97].

Um die Szenen für die Analyse einer bestimmten geographischen Region nutzen zu können, werden aus jeder Szene, die einen Teil dieser Region beinhaltet, quadratische Teile der Originalszene ausgeschnitten. Diese ausgeschnittenen Bereiche der ursprünglichen Szene werden Kacheln genannt. Dann wird für jeden Pixel der Kachel die Zugehörigkeit der mit dem Pixel assoziierten Koordinate zum Zielgebiet geprüft. Wenn ein Pixel relevant ist, wird er anhand seiner, aus der Position des Satelliten zum Aufnahmezeitpunkt ermittelten, Position in einem finalen Bild hinzugefügt.

Die Aufbereitung von Satellitenbildern muss vor einer wissenschaftlichen Analyse erfolgen, damit die Szenen unabhängig von Witterungseinflüssen, Atmosphäreninterferenzen, Fehlpositionierungen und sonstiger Störfaktoren untersucht werden können. Durch die zunehmend bessere Qualität von Satellitenbildern, die durch Fernerkundungssatelliten aufgezeichnet werden [MSWI04], können detailliertere Analysen getätigt werden. Jedoch steigt mit zunehmender Größe der Bilddateien auch der Rechenaufwand, um die Szenen aufzubereiten und zu analysieren. Mit zunehmender Datenmenge wird eine massiv parallelisierbare Vorgehensweise bei der Aufbereitung und der Analyse von Satellitenbildern attraktiver. Denn verteilte Systeme lassen sich meist kostengünstiger und flexibler erweitern als einzelne Maschinen, so dass das System bei einer unerwartet großen Datenmenge schnell erweitert werden kann. Dadurch lässt sich eine schnellere Ausführung der Prozesse erreichen.

2.2 Parallele Datenverarbeitungssysteme

Seit mehreren Jahren ist ein massiver Anstieg der global produzierten Datenmengen zu beobachten [EMC14]. Diese Menge an Daten ist mithilfe traditioneller Methoden der sequentiellen, stapelweisen Datenverarbeitung nicht effizient zu verarbeiten. Aus diesem Grund wird eine verteilte Verarbeitung von Daten in vielen Bereichen zunehmend populär. Dies gilt insbesondere für Daten, die gemäß der in Sektion 2.2.1 beschriebenen Kriterien als Big Data klassifiziert werden. Um eine parallele Verarbeitung von Big Data zu ermöglichen, wurden bestehende parallele Datenverarbeitungsmechanismen erweitert. Insbesondere das Map-Reduce System [DG04] bewirkte

eine grundlegende Veränderung bei der Vorgehensweise zur Verarbeitung großer Datenmengen. In der Folge wurde Map-Reduce erweitert und flexibler einsetzbar. Diese Entwicklung wird in [Sektion 2.2.2](#) erläutert. Eines der Systeme auf Basis von Map-Reduce ist Apache Flink [[Foud](#)]. Es ermöglicht eine massiv parallelisierbare und echtzeitnahe Verarbeitung von großen Datenmengen. Die konzeptionelle Struktur von Apache Flink wird in der [Sektion 2.2.3](#) beschrieben. Als Alternative zu parallelen Ansätzen existiert die bisherigen sequentiellen bzw. händisch parallelisierbaren Ansatz. Dieser wird am Beispiel der Programmiersprache Python in [Sektion 2.2.4](#) kurz beschrieben. Um die Ausführung von Algorithmen auf verschiedenen Systemen bewerten und vergleichen zu können, müssen vergleichende Metriken genutzt werden, die in [Sektion 2.2.5](#) kurz eingeführt und beschrieben werden.

2.2.1 Bedeutung und Eigenschaften von Big Data

Für das Jahr 2020 wird in der Folge der weltweit zunehmenden Generierung von Daten ein weltweites Datenaufkommen von 44 Zettabyte prognostiziert [[EMC14](#)]. Zusätzlich zu dieser schnell wachsenden Menge an verfügbaren Daten wächst auch der Bedarf diese nutzbringend zu analysieren. Insbesondere Forschungseinrichtungen und Unternehmen verfügen über immer größere Datenmengen und versuchen Erkenntnisse aus diesen zu gewinnen. Ein weiterer Teil dieser Daten wird durch die zunehmende Verbreitung des Internets der Dinge und die zunehmende Nutzung von Internetdiensten durch Konsumenten generiert. Traditionelle Methoden der Datenanalyse reichen jedoch nicht mehr aus, um diese Daten auszuwerten.

Dies resultiert aus den vier Eigenschaften, durch die Big Data definiert werden. Insbesondere die drei Charakteristika Volumen (engl. *volume*), Komplexität (engl. *variety*) sowie die echtzeitnahe Verfügbarkeit und schnelle Verarbeitung (engl. *velocity*) von Daten, die bereits 2001 von Dick Laney [[Lan01](#)] beschrieben wurden, erschweren die Verarbeitung mithilfe traditioneller Datenverarbeitungsmethoden. So können beispielsweise relationale Datenbanken unstrukturierte Daten nicht selbstständig kategorisieren bzw. strukturieren, da sie lediglich für die Verarbeitung von bereits strukturierten Daten konzipiert wurden. Hinzu kommt die nicht garantierte Zuverlässigkeit und Einheitlichkeit der Daten (engl. *veracity*) [[ZdP⁺12](#)], die eine Strukturierung der Daten nach festen Mustern erschweren können. Im folgenden werden die vier Eigenschaften kurz erläutert.

Volume. Im Rahmen des generellen Anstiegs von zu verarbeitenden Datenmengen müssen Datenverarbeitungssysteme zunehmend mit großen Datenmengen umgehen. Dadurch entstehen neue Anforderungen bei der Speicherung und Verarbeitung der Daten. Zunehmend sind dabei einzelne, große Datenmengen von Bedeutung. Beispiele dafür sind unter anderem das Sloan Digital Sky Survey, das seit 1998 insgesamt 116 Terabyte an astronomischen Daten gesammelt hat [[YAJEA⁺00](#), [AAP⁺15](#)] und das 1000 Genomes Project [[Bak10](#)], das zwischen 2008 und 2013 insgesamt 464 Terabyte Daten zum menschlichen Genom sammelte. Weitere Beispiele sind das CERN, dessen Large Hadron Collider täglich circa 1 Petabyte Daten produziert, und Google, das bereits im Jahr 2008 rund 20 Petabyte Daten pro Tag verarbeitete [[DG08](#)]. [Evtl. ein oder zwei Beispiele weniger nehmen. Streichkandidaten: SDSS, 1000G.](#)

Variety. Gesammelte Daten weisen vielfältige Datenstrukturen auf. Es werden nicht-strukturierte, semistrukturierte sowie strukturierte Daten gesammelt. Außerdem ist eine vorliegende Datenstruktur aufgrund von Inkompatibilität mit anderen Datenstrukturen möglicherweise schwierig in Bezug zu anderen Daten zu bringen. Ein Grund dafür ist der massive Anstieg an unterschiedlichen Datenquellen, deren erhobenen Daten nicht immer aufeinander abgestimmt sind. Daraus können sich Herausforderungen bei der Normierung von Daten ergeben. Denn Daten müssen teilweise selbstständig kategorisiert werden, oder komplett unstrukturiert gespeichert werden.

Velocity. Anwendungsfälle, die eine echtzeitnahe Verarbeitung von großen Datenmengen fordern, werden immer zahlreicher. Diese Verarbeitungsgeschwindigkeit ist aber nur umzusetzen, wenn die Datenverarbeitungssysteme mithilfe parallelisierter Architekturen auf eben solche ausgelegt

sind, da die Daten teilweise sehr schnell verfügbar sein müssen. Ein Beispiel sind autonom steuernde Fahrzeuge, die nur bei sofortiger und schneller Auswertung von Sensordaten angemessen auf durch Sensoren ermittelte Hindernisse reagieren können. Hinzu kommen Anwendungsszenarien, bei denen zusätzlich ein hoher Datendurchsatz erforderlich ist.

Veracity. Gesammelten Daten sind weder garantiert korrekt noch garantiert komplett. Durch falsche Modellannahmen oder hohe Latenzen einiger Datenquellen kann es zu weiteren Unsicherheiten bezüglich der Validität der Daten kommen. Big Data sind folglich immer möglicherweise fehlerbehaftet. Analysesysteme müssen auf diesen Umstand insofern reagieren, dass nicht valide Daten automatisiert erkannt und aus der Analyse ausgenommen werden. [Beispiel](#) Aufgrund dieser Eigenschaften mussten in den letzten Jahren immer wieder neue Konzepte und Systeme entwickelt werden, um Big Data verarbeiten zu können.

2.2.2 Systeme zur massiv parallelen Datenverarbeitung

In Anbetracht des steigenden Bedarfs an Techniken, mit deren Hilfe Big Data verarbeitet werden können, wurden die Entwicklung neuer und die Weiterentwicklung bestehender Technologien und Konzepte im Bereich Big Data innerhalb der letzten Jahre vorangetrieben. Dazu zählen insbesondere massiv parallelisierbare Rechnerstrukturen in Verbindung mit neuartigen Datenverarbeitungssystemen, die diese Konzepte und Technologien verwenden um Big Data verarbeiten zu können.

Die Entwicklung paralleler Rechnerstrukturen begann in den 1970er Jahren im Rahmen der Konstruktion von Computern mit mehreren kleinen Prozessoren (*Computer with multiple mini-processors*) [Bel71, WB72]. Die Entwicklung nutzbarer parallel arbeitender Computer begann in den 1980er Jahren [Sei85]. Gleichzeitig wurden parallelisierte Algorithmen konzipiert und umgesetzt [BH85]. Seitdem schritt die Weiterentwicklung parallelisierter Architekturen und Konzepte mit steigendem Tempo fort [TW12]. Während früher einzelne Maschinen mit parallel geschalteten Komponenten zur Bearbeitung aufwändiger Datenverarbeitungsaufgaben eingesetzt wurden, werden aktuell vermehrt Computercluster eingesetzt. Diese bestehen aus mehreren Maschinen, die mithilfe eines losen Netzwerks verbunden sind und so einen virtuellen Supercomputer darstellen [HDF13]. Aufgrund der Beschaffenheit der Computercluster lässt sich die Anzahl an zusammengeschlossenen Maschinen flexibel definieren. Auf diese Weise kann die Rechenleistung eines solchen Netzwerks kontinuierlich an die Anforderungen angepasst werden. Dies schafft optimale Voraussetzungen für die Verarbeitung von Big Data, da parallelisierte Strukturen einfacher erweitert werden können. So lässt sich die Kapazität des Clusters an den Umfang der zu analysierenden Daten anpassen. Einschränkend müssen aber auch die Grenzen von parallelisierten Verarbeitungsstrukturen berücksichtigt werden. Laut Amdahls Gesetz kann die Beschleunigung der Ausführungsgeschwindigkeit von Programmen durch eine parallele Ausführung maximal linear zur Anzahl der Prozessorkerne ansteigen [Amd67]. Dies resultiert aus Programmteilen, die zwangsweise sequentiell durchgeführt werden müssen. Jedes Programm besitzt solche Programmteile, etwa Speicherallokationen oder der sequentielle bzw. synchrone Zugriff von Threads auf geteilte Ressourcen. [Gustafsons Gesetz notwendig?](#)

Für eine effiziente Nutzung physischer paralleler Strukturen müssen parallelisierbare Algorithmen verwendet werden. Ein prägendes System, das die Implementierung solcher Algorithmen ermöglicht, ist das 2004 veröffentlichte Map-Reduce System [DG04]. Inspiriert durch ein ähnliches Konzept aus der funktionalen Programmierung ermöglicht es die nebenläufige Berechnung von großen Datenmengen. Darüber hinaus bietet es eine selbstständige Korrektur von bei Ausfällen von Netzwerkknoten verlorenen Daten. Dabei nutzt es ein verteiltes Dateisystem, wie beispielsweise das Google File System [GGL03], das auf einem Computercluster ausgeführt wird.

Jedes Map-Reduce Programm nutzt die zwei Funktionen zweiter Ordnung *map* und *reduce*. Für diese müssen vom User jeweils eine *User-defined function* implementiert werden. Das Map-Reduce

System parallelisiert dann die Funktionen zweiter Ordnung map und reduce auf Teilmengen der zu verarbeitende Datenmenge und verarbeitet diese Daten gemäß der spezifizierten UDFs. Wichtig ist dabei die zu berücksichtigende Struktur des Programms, die genau eine map-Funktion sowie genau eine auf die map-Funktion folgende reduce-Funktion voraussetzt.

Bei der in Abbildung 2.1 abgebildeten modellhaften Ausführung eines Map-Reduce Programms muss zunächst sichergestellt werden, dass die zu verarbeitenden Daten auf eine Weise partitioniert sind, die die Verarbeitung mittels Map-Reduce erlaubt. Dies wird durch eine Partitionierung der Daten in sogenannte *chunks* gewährleistet. Außerdem müssen die map- und die reduce-Funktion auf die Netzwerkknoten kopiert werden. Diese Arbeitsschritte sind in der Grafik als Schritt 1 bezeichnet. Dann werden die Netzwerkknoten des genutzten Clusters gemäß einer Master-Worker Architektur initialisiert. Zunächst wird in Schritt 2 ein Master-Knoten bestimmt. Der Master-Knoten weist den unterschiedlichen Worker-Knoten map- bzw. reduce-Aufgaben zu. Zusätzlich erhält der Worker-Knoten eine Referenz auf den Speicherort der zur verarbeitenden Daten. Des weiteren koordiniert der Master-Knoten die Worker und prüft sie in regelmäßigen Abständen auf korrekte Funktionalität. Sollte einer der Worker-Knoten nicht mehr funktionsfähig sein, wird er aus dem restlichen Verarbeitungsprozess ausgeschlossen. Die Aufgaben, die dem Knoten zugewiesen worden sind oder waren werden erneut als unerledigt gekennzeichnet und von anderen Workern nochmals ausgeführt.

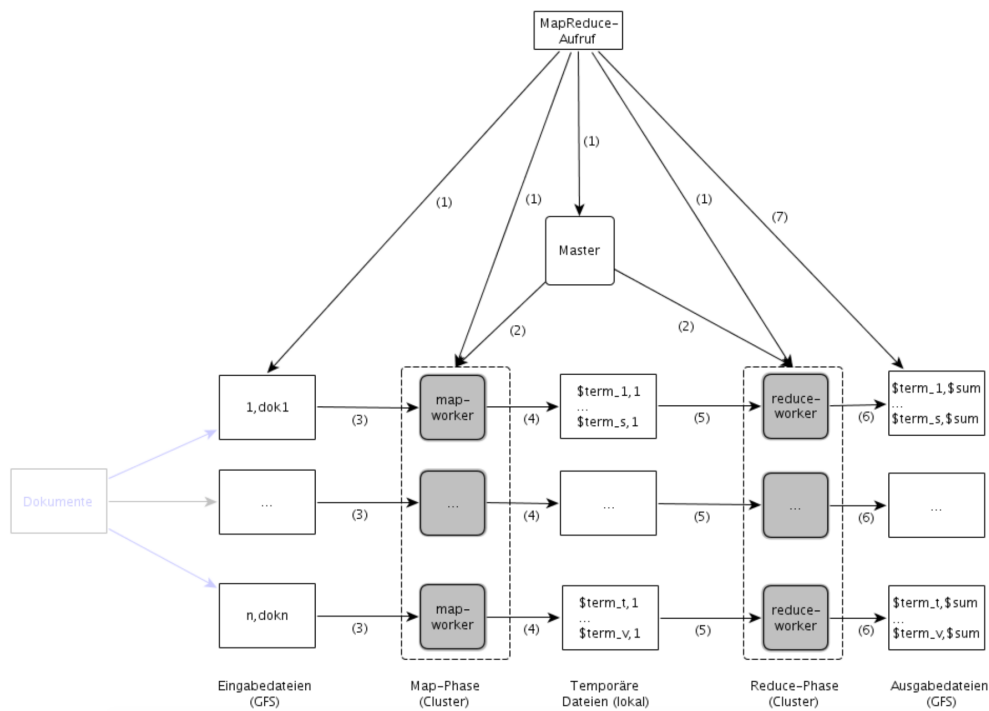


Abbildung 2.1: Systemübersicht eines Map-Reduce System [BR09]

Weiterhin ist auf Abbildung 2.1 in Schritt 3 zu erkennen, dass jeder Worker, der eine Map-Aufgabe ausführt, eine Teilmenge der zu verarbeitenden Gesamtdatenmenge als Eingabe erhält. Die Daten müssen dazu in Form von Schlüssel-Wert Paaren gespeichert sein. Diese werden dann gemäß der vom Nutzer spezifizierten UDF verarbeitet. Als Ausgabe werden keine, ein oder mehrere Schlüssel-Wert Paare durch die map-Funktion produziert und im lokalen Festpeicher der jeweiligen

Worker gespeichert. Aufgrund der Aufteilung der Gesamtdatenmenge in kleinere, von einander unabhängig prozessierbare Mengen, kann die Map-Phase parallelisiert ablaufen. Dabei werden auf jedem Worker des Clusters verschiedene Teilmengen der Eingabedaten verarbeitet.

Nach Beendigung der Map-Phase folgt die Shuffle-Phase, auch Repartitionsphase genannt. Diese ist in Abbildung 2.1 als Schritt 4 gekennzeichnet. Während dieser werden die durch die Map-Funktion produzierten Schlüssel-Wert Paare gemäß ihrer Schlüssel vorerst partiell gruppiert. Wenn der Speicherbedarf für alle Schlüssel-Wert Paare einer Gruppe größer ist als der verfügbare Speicher auf einem Worker werden mehrere partielle Gruppen mit demselben Schlüsselwert gebildet. Jede partielle Gruppe wird anschließend in Schritt 5 auf verfügbare Worker übertragen und dort zu einer globalen Gruppe zusammen gefügt. *Ist wirklich Phase 5 die Shuffle Phase? Ich lese meine Quelle so, dass Phase 5 die Verteilung der repartitionierten Gruppen an die Worker ist. Quelle: <http://www2.informatik.hu-berlin.de/~bethge/public/MapReduce/document.pdf>*

Die in Schritt 6 dargestellte reduce-Funktion wird dann auf jede in der Shuffle-Phase erzeugte globale Gruppe von Schlüssel-Wert Paaren angewendet. Die spezifizierte UDF wird auf die Werte aller Schlüssel-Wert Paare einer Gruppe angewendet. Als Ausgabe wird entweder keine, ein oder mehrere Schlüssel-Wert Paare produziert.

Die Gesamtausgabe des Map-Reduce Programms besteht aus allen durch reduce-Funktionen produzierten Schlüssel-Wert Paaren.

Aufgrund der sehr genau festgelegten Programmstruktur, die zur Nutzung von Map-Reduce eingehalten werden muss wird der Anwender gezwungen einen Datenverarbeitungsprozess auf mehrere aufeinanderfolgende Map-Reduce Instanzen zu verteilen, wenn dieser zu komplex ist.

Seit seiner Einführung im Jahr 2004 wurde das Map-Reduce Paradigma erweitert und bietet nun mehr Flexibilität, da es um weitere Funktionen zweiter Ordnung ergänzt wurde, die zusätzliche Datentransformationen ermöglichen. Aufgrund des Einsatzes von Map-Reduce in Hadoop [Foue] und anderen Systemen wie beispielsweise Spark [ZCF⁺10] bleibt es aber eines der dominanten Paradigmen bei der Verarbeitung großer Datenmengen. Auch Apache Flink [Foud] nutzt eine ähnliche Funktionsweise, um die parallelisierten Datenströme zu verwalten.

Neben den bereits erwähnten Systemen Apache Hadoop und Apache Flink existieren weitere populäre Systeme, beispielsweise Apache Spark [Foub, ZCF⁺10], Apache Storm [Fouc, Jon13] oder Asterix [Ast, AKL⁺12]. Diese unterscheiden sich bereits in ihrer Architektur. Während Apache Hadoop das Map-Reduce Paradigma sehr fixiert umsetzt und somit dem Nutzer nur eingeschränkte Mittel zur Spezifizierung seines Programmes lässt, setzen Systeme wie Apache Spark und Apache Flink das Map-Reduce Paradigma flexibler ein. Nachfolgend wird ein kurzer Überblick über die Funktionsweise der genannten Systeme gegeben.

Apache Hadoop. Apache Hadoop implementiert das Map-Reduce System im Rahmen eines freien Projekts. Dabei können sowohl die Map- als auch die Shuffle- und Reducephase vom Nutzer individuell konfiguriert werden. Als freier Ersatz für GFS nutzt Hadoop das Dateisystem Hadoop Distributed File System (HDFS). In der Basisversion ermöglicht Hadoop lediglich Stapelverarbeitungen, echtzeitnahe Datenverarbeitung ist nicht vorgesehen. Hadoop wird in diversen Firmen genutzt, zum Beispiel verwendet Facebook das System zur Analyse von Datenmengen von bis zu 100 Petabyte [Bor13].

Apache Spark. Apache Spark ist eine allgemeine Datenverarbeitungsplattform. Es beinhaltet mehr Funktionen zweiter Ordnung als Hadoop, es gibt also mehr Möglichkeiten verschiedene Datentransformationen auszuführen. Über den Funktionsumfang von Hadoop hinaus ermöglicht es neben der Stapelverarbeitung weitere Anwendungen wie zum Beispiel maschinelles Lernen. Des weiteren ist ebenfalls möglich Spark mit anderen Bibliotheken zu ergänzen. Diese Eigenschaften machen Spark gegenüber Hadoop für viele Anwendungszecke überlegen.

Apache Storm. Apache Storm ist ein Datenverarbeitungssystem, das speziell auf die schnelle echtzeitnahe Verarbeitung von Daten ausgelegt ist. Ein Storm-Programm hat die Form eines direkten, azyklischen Graphs, dessen Kanten Datenströme darstellen. Ein Storm-Programm kann als

datentransformierende Pipeline bezeichnet werden. Ein Storm-Programm entspricht einer Implementierung eines abstrakten Map-Reduce-Jobs. Im Unterschied zu einem Map-Reduce Job läuft ein Storm-Programm, bis es beendet wird. Dies unterstreicht den Einsatzzweck für echtzeitnahe Verarbeitung von Daten, die ohne Zeitbegrenzung generiert werden.

AsterixDB. AsterixDB ist ein *Big Data Management System*, dass seit 2009 von den Universitäten Irvine, San Diego und Riverside gemeinschaftlich entwickelt wird. Das Ziel von AsterixDB ist eine effiziente parallelisierte Ausführung von Analysen auf semistrukturierten Daten. Dabei wird versucht Datenbanktechnologie mit Big-Data-Technologie zu kombinieren, um große Daten effizient verarbeiten zu können.

2.2.3 Apache Flink

Ein Datenverarbeitungssystem, das auf eine massiv parallelisierte Verarbeitung von großen Datenmengen spezialisiert ist, ist Apache Flink. Es ging 2014 aus Stratosphere hervor [Bra], einem Forschungsprojekt, das seit 2010 kooperativ von Forschern verschiedener Universitäten entwickelt wurde [BEH⁺10, ABE⁺14]. Seit Januar 2015 ist Apache Flink ein Top-Level Projekt der Apache Software Foundation [Fou15]. Ähnlich wie andere Systeme ermöglicht es Apache Flink bereits vorhandene Daten in einem Stapel-Verfahren zu analysieren. Darüber hinaus können jedoch auch in Echtzeit zu verarbeitende Daten im Rahmen eines *Streaming*-Verfahrens prozessiert werden.

Die Hauptkomponenten von Apache Flink sind die Flink-Laufzeitumgebung (engl. *Flink Runtime*) und der Flink-Optimierer (engl. *Flink Optimizer*). Darüber hinaus verfügt Flink über Programmierschnittstellen für die Hochsprachen Java, Python und Scala, mithilfe derer Nutzer Flink-Programme spezifizieren können.

Flink implementiert das vom Map-Reduce System bekannte Schema von Funktionen zweiter Ordnung und gekapselten nutzerdefinierten Funktionen erster Ordnung. Dabei beschreibt die Funktion zweiter Ordnung die Art der Datentransformation, die UDF definiert die vom Nutzer spezifizierte Verarbeitungslogik, die auf die Daten angewendet werden soll. Im Gegensatz zum Map-Reduce System verfügt Flink jedoch nicht nur über die Funktionen zweiter Ordnung Map und Reduce, sondern bietet insgesamt 15 verschiedene Datentransformationen. Diese werden als Operatoren bezeichnet. Weitere wichtige Operatoren sind neben Map und Reduce beispielsweise Cross, Match und CoGroup, die in der Grafik 2.2 dargestellt sind. Die formalen Definitionen der Operatoren sind in Tabelle 2.1 aufgeführt. Nachfolgend werden die genannten Operatoren kurz beschrieben. Der *Map*-Operator besitzt eine Eingabemenge und erstellt für jedes Element dieser Menge eine eigene Gruppe, die genau dieses Element enthält. Der *Reduce*-Operator erstellt für jeden vorhandenen einzigartigen Schlüsselwert eine Gruppe und gruppiert alle Einträge mit demselben Schlüssel in dieselbe Gruppe ein. Der *Cross*-Operator erstellt für jedes Eingabepaar von Werten aus zwei Eingabemengen eine Gruppe, die das kartesische Produkt der beiden Werte darstellt. Der *Match*-Operator erstellt eine Gruppe für jedes Eingabepaar von Werten zweier Eingabemengen, genau dann wenn beide Werte denselben Schlüsselwert besitzen. Der *CoGroup*-Operator erstellt für jeden einzigartigen Schlüsselwert, der in mindestens einer der beiden Eingabemengen enthalten ist, eine Gruppe und gruppiert in dieser alle Werte, die diesen Schlüsselwert aufweisen. Jeder Operator besteht also aus jeweils einer spezifischen Transformation sowie einer eingebetteten UDF. Zusätzlich besitzen Operatoren eine Eingabe- sowie eine Ausgabedatenmenge in Form eines Datenstroms. Ein Flink-Programm wird als ein Datenfluss in Form eines direkten und azyklischen Graphen dargestellt, dessen Knoten Operatoren und dessen Kanten Datenströme darstellen. [Ref auf Abbildung eines Beispieldatenstroms.](#) Dieser wird *Operator DAG* genannt.

Der Flink-Optimierer erhält den mithilfe einer Programmierschnittstelle spezifizierten Operator DAG als Eingabe. Die Aufgabe des Flink-Optimierers ist die Umwandlung des Operator DAG in einen parallelisiert ausführbaren Datenfluss, den *OptimizedPlan*. Dieser Datenfluss soll möglichst stark parallelisierbar sein, damit eine effiziente Verarbeitung durch die Flink-Laufzeitumgebung

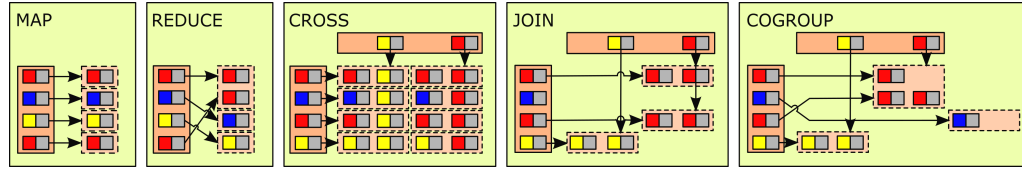


Abbildung 2.2: Funktionen zweiter Ordnung: (a) Map, (b) Reduce, (c) Cross, (d) Match und (e) CoGroup [HPS⁺12]

Operator	Formale Definition
Map	Map: $R \times f \rightarrow [f(r_1), \dots, f(r_i), \dots, f(r_N)]$
Reduce	Reduce: $R \times f \times K \rightarrow [f(r_1^{k_1}, \dots, r_{n_1}^{k_1}), \dots, f(r_i), \dots, f(r_N)]$ mit K als Set von Schlüsseln $\{k_1, \dots, k_l\}$ in R
Cross	Cross: $R \times S \times f \rightarrow [f(r_1, s_1), f(r_1, s_2), \dots, f(r_N, s_M)]$
Match	$R \times S \times K \times F \times f \rightarrow [\{f(r, s) r.K = s.F\}]$ mit K und F als Schlüssel für R und S .
CoGroup	$R \times S \times K \times F \times f \rightarrow [f(r_1^{v_1}, \dots, r_{n_1}^{v_1}, s_1^{v_1}, \dots, s_{m_1}^{v_1}), \dots, f(r_1^{v_l}, \dots, r_{n_l}^{v_l}, s_1^{v_l}, \dots, s_{m_l}^{v_l})]$ mit $\{v_1, \dots, v_l\}$ als aktive Domain von K und F

Tabelle 2.1: Formale Definition der Operatoren Map, Reduce, Cross, Match und CoGroup mit den Eingabedatenmengen $R = [r_1, \dots, r_i, \dots, r_N]$, $S = [s_1, \dots, s_i, \dots, s_N]$ und f als UDF für die jeweilige Transformation. [HPS⁺12]

ermöglicht werden kann. Um einen solchen Datenfluss zu erhalten werden abhängig von den verwendeten Operatoren programmspezifische Optimierungen vorgenommen. Dabei wird für jeden Operator auf Basis seiner Eingabedaten festgelegt, mit welcher internen Implementierung der Operator ausgeführt wird. Ein Beispiel dafür wäre die Ausführung eines Join-Operators wahlweise als Sort-Merge-Join oder als Hash-Join. Weitere Optimierungen umfassen die Wahl des Datenaustauschs zwischen den Operatoren sowie die Auswahl der Programmpunkte, an denen Zwischenergebnisse gespeichert werden. Außerdem wird versucht Daten, die durch einen Operator sortiert oder partitioniert wurden, mit derselben Sortierung weiterzuverwenden, um weitere Sortier- oder Partitionierungsprozesse zu vermeiden. Ist der optimale Datenflussgraph ermittelt, wird er zu einem *JobGraph*, einem direkten azyklischen Graphen bestehend aus Operatoren und Zwischenergebnissen als Knoten, umgewandelt. Zur Ausführung durch die Flink-Runtime wird dieser Graph in einen *ExecutionGraph* umgewandelt. Ein Beispiel für einen JobGraph sowie einen ExecutionGraph wird in Abbildung 2.3 gezeigt.

Flink nutzt ebenso wie das Map-Reduce System eine Master-Worker Architektur zur Organisation des Netzwerkes, auf dem das Flink-Programm ausgeführt wird. Dementsprechend verwaltet der Master-Knoten, auch *Jobmanager* genannt, eine variable Anzahl an Worker-Knoten, *TaskManager* genannt. Diese TaskManager verfügen jeweils über eine variable Anzahl von *TaskSlots*, die die Anzahl der parallel ausführbaren Prozesse festlegt. Analog zum Map-Reduce-System koordiniert der JobManager die Zuweisung der zu bearbeitenden Teilaufgabe des ExecutionGraph an die assoziierten TaskManager, die diese dann ausführen. Darüber hinaus werden die Funktionsfähigkeit der TaskManager sowie der Status jeder auszuführenden Teilaufgabe durch den JobManager überwacht. Im Falle eines Ausfalls eines TaskManagers werden dessen Aufgaben von anderen Worker-Knoten übernommen bzw. wiederholt. Im Gegensatz zum ursprünglichen Map-Reduce-System ist ein Datenaustausch zwischen den TaskManagern in Form von *Data Streams* möglich.

Bei der Ausführung eines Flink-Programms wird nach der Optimierung des Operator DAGs das System gemäß nutzerdefinierter Prämissen wie zum Beispiel dem Grad der Parallelität initiali-



Abbildung 2.3: Modell eines Jobgraphs und des dazugehörigen ExecutionGraphs [Foua]

siert. Der JobManager erhält den vom Flink-Optimizer generierten JobGraph als Eingabe. Dieser wird, wie in Abbildung 2.3 dargestellt, in einen parallel strukturierten *ExecutionGraph* transformiert. Für jeden Knoten des Jobgraphs wird ein *ExecutionJobVertex* erstellt. Dieser verantwortet die Ausführung des durch den Knoten repräsentierten Operators sowie die Weitergabe der, aus der Anwendung des Operators auf die Eingabedaten resultierenden, Zwischenergebnisse. Dabei werden n *ExecutionVertices* genutzt, wobei n dem vom Nutzer festgelegten Grad der Parallelität entspricht. Jeder *ExecutionVertex* übernimmt die Ausführung einer der parallelisierten Verarbeitungsinstanzen des Operators. Dazu wird ein vom JobManager bestimmter Worker-Knoten genutzt. Der *ExecutionJobVertex* garantiert die komplette Ausführung des Operators auf der gesamten Eingabedatenmenge, indem der Status jedes *ExecutionVertex* verfolgt wird. Sollte einer der *ExecutionVertices* ausfallen wird die gerade verarbeitete Ausführungsinstanz an andere *ExecutionVertices* delegiert. Darüber hinaus beinhaltet der *ExecutionGraph* die Zwischenergebnisse, die zwischen der Anwendung verschiedener Operatoren erstellt werden. Dadurch ist jederzeit bekannt, ob die Bedingungen für die Ausführung eines weiteren Operators gegeben ist, oder ob noch nicht alle Eingabedaten in Form von Zwischenergebnissen verfügbar sind.

2.2.4 Python

Python ist eine höhere, interpretierte Programmiersprache, die seit 1989 existiert, quelloffen ist und fortwährend weiter entwickelt wird. Obwohl Python eine imperative Sprache ist, können auch objektorientierte und funktionale Programmierparadigmen verwendet werden. Jedes Python-Programm wird bei der Ausführung mittels des Python-Interpreters in Bytecode umgewandelt. Dieser ist plattformunabhängig lauffähig, so dass Python-Programme ohne weitere Modifizierung auf mehreren Systemen lauffähig sind. Eine weitere Eigenschaft von Python sind die sehr umfangreiche Standardbibliothek sowie die Erweiterbarkeit um Module und Bibliotheken. Des weiteren gibt es mit der Python-API die Möglichkeit Python-Code durch C- beziehungsweise C++-Bibliotheken

zu erweitern [Mar06]. Unter anderem aufgrund der maschinennahen Umsetzung und der effizient implementierten Compiler von C beziehungsweise C++ besitzen diese Sprachen einen Leistungsvorteil gegenüber anderen höheren Programmiersprachen. Dies kann in Python genutzt werden, indem leistungsintensive Programmteile direkt als C- oder C++-Code eingebunden und ausgeführt werden. Ein weiterer Vorteil von Python ist die starke Typisierung und daraus resultierend die bessere Optimierung des Codes bei der Übersetzung durch den Python-Interpreter. Zusätzlich nutzt Python Duck-Typing, also die Typbeschreibung eines Objekts alleinig durch das Vorhandensein bestimmter Methoden beziehungsweise Attribute, was die Nutzbarkeit der Typisierung durch den Entwickler vereinfacht.

Ein Schwachpunkt von Python im Bezug auf die schnelle Verarbeitung großer Datenmengen ist die nicht auf automatisierte Parallelisierung ausgelegte Architektur. Daraus resultiert eine unzureichende Skalierbarkeit der Datenverarbeitung, sobald Daten, deren Größe die Arbeitsspeichergröße der ausführenden Maschine übersteigt, verarbeitet werden müssen. Dies ist insbesondere im Zuge der in Abschnitt 2.2.2 aufgezeigten Entwicklung hin zu größeren zu verarbeitenden Datenmengen relevant. Entwicklungen, die eine bessere Parallelisierung und damit einhergehend eine bessere Skalierbarkeit von Python Projekten zum Ziel haben, sind jedoch meist nicht universal anwendbar oder bieten nicht den Funktionsumfang von Big-Data Systemen [Van15, Pem15]. Die Funktionalität dieser Entwicklungen wird dabei durch Module bereitgestellt, da sie nicht als Grundfunktion in Python existiert sondern projektweise hinzugefügt werden muss. Dies erhöht den zu leistenden Anpassungs- und Entwicklungsaufwand im Vergleich zu anderen Systemen, die mit Rücksicht auf diese Anwendungsfälle konzipiert wurden.

Trotzdem nutzen viele Anwender, die Big-Data Anwendungen entwickeln, Python [MA11]. Einerseits ist die Sprache ein seit vielen Jahren bewährtes Werkzeug [Bea00, Oli07], das durch Nutzung von Bibliotheken wie SciPy und NumPy effizient implementierbare Algorithmen ermöglicht, ohne dass alle genutzten Funktionen selbst programmiert werden müssen. Weitere Gründe für das Festhalten an Python sind zum Beispiel die Möglichkeit zur Einbindung von Fortran und die bei Nutzung neuerer Systeme notwendige Reimplementation bereits existierender Lösungen. Die Kosten-Nutzen-Relation bei einer Nutzung anderer Systeme wie zum Beispiel Apache Flink muss aufgrund des nötigen Entwicklungsaufwands projektabhängig und individuell aufgestellt werden.

2.2.5 Vergleichsmetriken

Um die Leistungsfähigkeit der verschiedenen Systeme beziehungsweise Programmiersprachen vergleichen zu können, müssen Vergleichsmetriken definiert werden. Diese umfassen verschiedene Leistungskriterien, hinsichtlich derer die Systeme verglichen werden. Kriterien sind zum Beispiel die Datenverarbeitungsgeschwindigkeit, die Skalierbarkeit sowie das Verhältnis vom Preis zur Performance einer Anwendung. Einerseits kann eine offene Messung ohne Bezugswerte vorgenommen werden, die alle gemessenen Systeme untereinander vergleicht. Andererseits können Systeme hinsichtlich eines Referenzsystems evaluiert werden. Um die zu vergleichenden Kriterien mehrerer Varianten einer Anwendung auf verschiedener Systemen mit einem Referenzsystem vergleichen zu können, muss für jedes dieser Kriterien ein Referenzwert des Referenzsystems existieren. Dieser fungiert als Vergleichswert für alle Werte, die für die zu vergleichenden Anwendungsvarianten hinsichtlich dieses Kriteriums gemessen werden. Anhand der Unterschiedlichkeit von Mess- und Bezugswert kann das zu evaluierende System mit dem Bezugssystem verglichen werden. Hauptkriterium für traditionelle Datenbanksysteme ist die Datenverarbeitungsgeschwindigkeit, die als Durchsatzmetrik in Form von Arbeit pro Zeit definiert ist [Gra92]. Für diese Evaluation werden standardisierte Benchmarks verwendet, etwa TPC-H. Für Big-Data Anwendungen existiert darüber hinaus ein weiteres wichtiges Leistungskriterium, die Skalierbarkeit des Systems. Bis zum aktuellen Zeitpunkt hat sich allerdings noch kein standardisierter Benchmark für Big-Data Systeme etabliert. Jedoch werden bereits seit mehreren Jahren die Entwicklung solcher Benchmarks

vorangetrieben [CRK14]. Ein Big-Data Benchmark muss dabei die Kriterien Datenverarbeitungsgeschwindigkeit, Skalierbarkeit sowie Preis/Performance evaluieren und auf sämtlichen Big-Data Systemen ausführbar sein [BBN⁺13].

Aufgrund des in Sektion 2.2.1 beschriebenen Notwendigkeiten für parallele Datenverarbeitung vergrößern sich auch die zu verarbeitenden Datenmengen kontinuierlich. Deshalb ist die Skalierbarkeit des datenverarbeitenden Systems für Big-Data Anwendungen besonders wichtig, damit das Datenverarbeitungssystem flexibel auf die zu verarbeitende Datenmenge angepasst werden kann. Aufgrund ihrer parallelisierten Architektur, die aus mehreren miteinander verbundenen Maschinen besteht, sind weitere Maschinen teilweise sehr einfach integrierbar. Sie müssen häufig lediglich an die bestehende Netzwerkinfrastruktur des Systems angeschlossen werden. Bei sequentiell arbeitenden Datenverarbeitungssystemen ist die Skalierbarkeit des Systems erschwert, da lediglich einzelne, geeignete Teilprozesse eines sequentiellen Programms manuell parallelisiert werden können. Darüber hinaus muss auch der Umgang mit Ausfällen der genutzten Hardware betrachtet werden. Im Idealfall wird der Ausfall von Netzwerkknoten wie bereits beim Map-Reduce System automatisiert isoliert und die Neuberechnung von Teilergebnissen auf ein notwendiges Minimum reduziert. Dieses Minimum entspricht dabei der Anzahl der notwendigen Neuberechnungen, die benötigt werden, um die Korrektheit der Ergebnisse zu garantieren. Das Ersetzen von fehlerhafter Hardware ist bei sequentiell arbeitenden Programmen meist nicht möglich, ohne die Anwendung abzubrechen. **Ist das richtig?**

Die Skalierbarkeit hat auch unmittelbare Folgen für die weiteren Bewertungskriterien, insbesondere für die absolute Datenverarbeitungsgeschwindigkeit einer Anwendung. Aufgrund der Definition der Datenverarbeitungsgeschwindigkeit durch das Verhältnis von geleisteter Berechnungsarbeit pro Zeiteinheit sinkt sie bei einer Erweiterung der Rechenkapazitäten, wenn die Anwendung skaliert. Denn dementsprechend steigt die geleistete Berechnungsarbeit pro Zeiteinheit, was gleichbedeutend zu einer Beschleunigung der Datenverarbeitung ist. Die absolute Ausführungszeit einer Anwendung ist dabei das Produkt der Datenmenge und der Datenverarbeitungsgeschwindigkeit. Wächst die zu verarbeitende Datenmenge an, steigt demzufolge auch der nötige Verarbeitungsaufwand. Unter der Annahme einer identischen Datenverarbeitungsgeschwindigkeit steigt in diesem Fall auch die Ausführungsgeschwindigkeit. Aufgrund der im Allgemeinen wachsenden zu verarbeitenden Datenmengen muss die zur Verfügung stehende Rechenleistung erhöht werden, um die Datenverarbeitungsgeschwindigkeit und damit die Ausführungszeit einer datenverarbeitenden Anwendung zu verringern. Die Skalierbarkeit eines Big-Data Systems ist also essentiell für eine Datenverarbeitung mit gleicher Ausführungszeit bei wachsenden Datenmengen.

Obgleich die beiden genannten Kriterien die Leistungsfähigkeit eines Big-Data Systems definieren, muss zusätzlich der Preis des Systems im Verhältnis zur erzielten Datenverarbeitungsgeschwindigkeit betrachtet werden. Denn andernfalls wäre ein Leistungsvergleich von verschiedenen Datenverarbeitungssystemen nicht machbar, da etwaige Nachteile eines Systems durch zusätzliche oder bessere und somit teurere Hardware ausgeglichen beziehungsweise verschleiert werden könnten. Aufgrund der Nutzung von Clustern von handelsüblicher Maschinen in verteilten Systemen sind diese meist günstiger als einzelne Maschinen die eine gleichwertige Rechenleistung aufweisen.

Kapitel 3

Beschreibung und Umsetzung des Algorithmus zur Analyse von Pixelzeitreihen

3.1 Beschreibung des Algorithmus zur Analyse von Pixelzeitreihen

Das Ziel des in dieser Bachelorarbeit implementierten und evaluierten Algorithmus ist die Erkennung einer unerwarteten Veränderung der bewaldeten Fläche einer geographischen Region. Der Grad der Bewaldung wird dabei durch einen Vegetationsindex dargestellt. Um eine Veränderung des Vegetationsindex zu messen, werden die durch Landsat Satelliten aufgezeichneten Szenen der Zielregion hinsichtlich des jeweiligen Vegetationsindex untersucht. Auf Basis dieser Analyse können dann zukünftige Werte prognostiziert werden und etwaige Abweichungen der zukünftig tatsächlich gemessenen Werte automatisiert ermittelt werden. Die zu diesem Zweck eingesetzte Analyse ist eine kombinierte Anwendung von *Support Vector Regression* (SVR) [BPP07] und der Methode der kleinsten Quadrate. Diese werden im Rahmen des *Continuous Monitoring of Forest Disturbance Algorithm* (CMFDA) [ZWO12] genutzt.

Der in Algorithmus 1 angegebene Algorithmus zur Analyse der Veränderung des Vegetationsindex benötigt eine Menge von Szenen, die mindestens einen Teil der Zielregion beinhalten, als Eingabe. Diese werden in Zeile 2 zunächst gemäß ihres Aufnahmedatums gruppiert. Mithilfe der Schleife in Zeile 6 wird aus jeder der gruppierten Szenen dann die exakte Zielregion ausgeschnitten. Dieser Schritt ist notwendig, da etwaige Zielregionen zumeist kleiner sind als eine originale Landsat-Szene oder nur ein Teil der Zielregion tatsächlich in der Ausgangsszene abgebildet ist. Der ausgeschnittene Bereich wird dann als Kachel bezeichnet. Um die Eigenschaften und Ordnung der Kacheln zu verdeutlichen, wird ein Würfel mit den Dimensionen Breite der Szene, Höhe der Szene und dem Aufnahmezeitraum der Szenen modelliert. Dieser besteht aus den einzelnen sortierten Szenen, so wie in Abbildung 3.1 abgebildet. Die x- und die y-Dimension des Würfels entsprechen genau der Breite und der Höhe der Zielregion, während die z-Dimension den Aufnahmezeitraum t beschreibt.

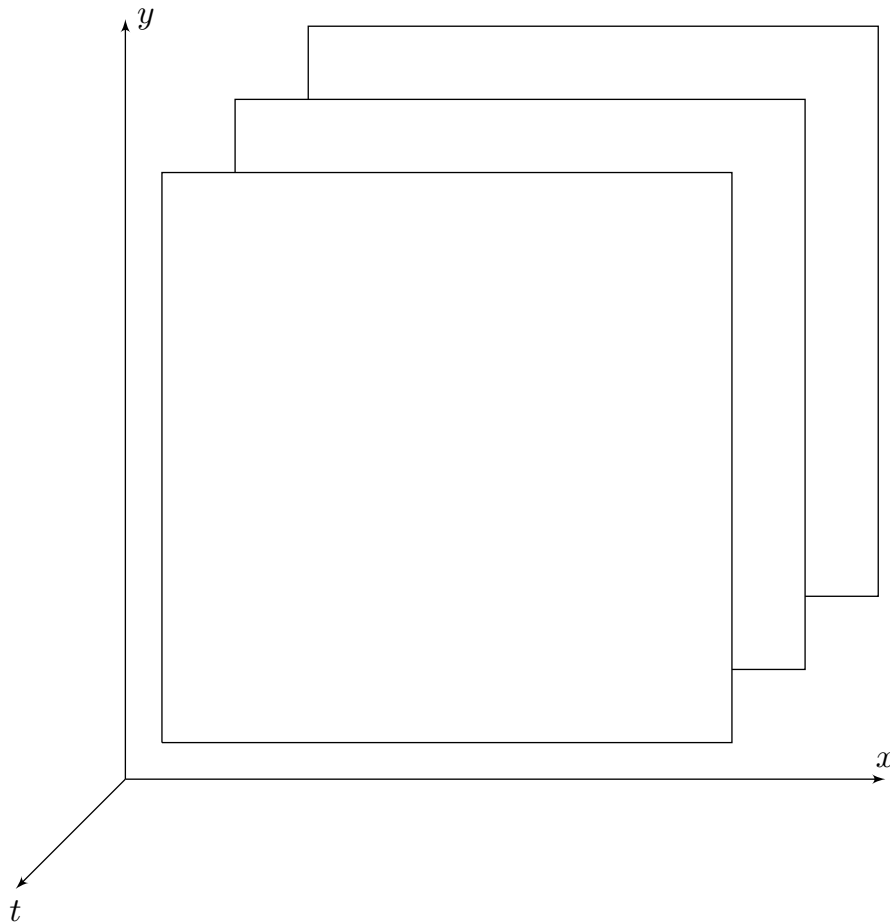


Abbildung 3.1: Beispielhafte Szenen derselben geographischen Region mit identischer Breite x und Höhe y sowie unterschiedlichem Aufnahmedatum t .

Jede Kachel dieses Datenwürfels muss nun in Zeile 6 in eine individuell festgelegte Anzahl an Teilkacheln zerschnitten werden. Dabei werden zunächst aus Gründen der Vereinfachung nur quadratische Teilkacheln betrachtet. Dies impliziert, dass die Breite und die Höhe der Zielregion einen gemeinsamen Teiler besitzen müssen. Theoretisch ist aber auch die Nutzung von allgemein rechteckigen Teilkacheln möglich. Generell gilt, dass alle Teilkacheln über identische Abmessungen verfügen müssen. Nachdem alle Kacheln in Teilkacheln geschnitten wurden, werden diese gemäß ihrer geographischen Position sowie des betrachteten Spektralbandes gruppiert. Die geographische Position ist in Abbildung 3.2 anhand der Tupel der Teilkacheln markiert. Durch die Gruppierung der Teilkacheln in Zeile 11 entstehen $n \times b$ verschiedene Gruppen mit $n = \{\text{Anzahl der Teilkacheln}\}$ und $b = \{\text{Anzahl der betrachteten Spektralbänder}\}$. Jede dieser Gruppen wird dann in Zeile 14 nach dem jeweiligen Aufnahmedatum der einzelnen Teilkacheln der Gruppe sortiert.

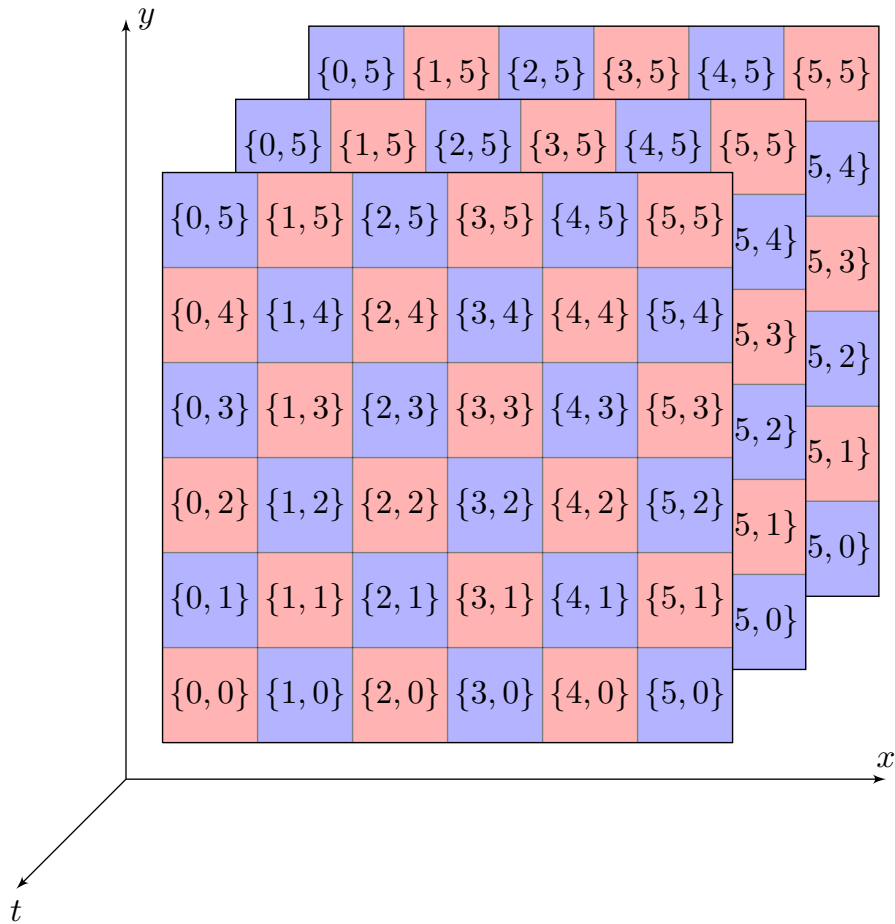


Abbildung 3.2: Die in Teilszenen gleicher Größe geschnitten Szenen s_i aus Abbildung 3.1. Die Tupel geben die Position der Teilszenen innerhalb der Szene an und werden als Gruppenschlüssel zum Gruppieren der Teilszenen verwendet.

Um die Veränderung des Vegetationsindex jedes Pixels analysieren zu können, muss in Zeile 16 zunächst für jeden Pixel eine Pixelzeitreihe gebildet werden. In dieser ist jedem Vegetationswert v_t genau ein fester Zeitpunkt t zugeordnet, zu dem der Wert durch die Auswertung des Pixelwerts des Satellitenbildes bekannt ist. Allerdings ist nicht jedem Wert t ein Wert v_t zugeordnet, da entweder keine Szene s_t analysiert wurde oder der Pixel in der Szene s_t keinen validen Wert enthält. Dies kann aufgrund externer Störfaktoren wie Wolken, Wolkenschatten oder Schnee zum Zeitpunkt der Aufnahme der Fall sein. Auf Basis der vorhandenen Werte wird dann in Zeile 16 mittels einer Regressionsanalyse eine Funktion approximiert, die die Werte für alle t bestmöglich annähert. Die Gesamtheit der ursprünglichen und der approximierten Werte wird dann genutzt, um in Zeile 16 auf dieselbe Weise zukünftige Werte des Vegetationsindex für jeden Pixel zu prognostizieren. Für beide Regressionsanalysen werden die Regressionsverfahren SVR und OLS eingesetzt, die nachfolgend erklärt werden.

Input: Set von Satellitenbildern im bsq-Dateiformat und zugehörigen Header Dateien S , Koordinaten der Zielbereichsbegrenzungen c_1 und c_2 , Grad der Zerschneidung der Szenen m_x, m_y

Data: $S := n$ Szenen s_i mit $0 \leq i \leq n$. c_1 und $c_2 :=$ Tupel aus Breiten- und Längengrad. m_x, m_y als Integer, des weiteren gilt (vorerst) $m_x = m_y$

Result: Noch festzulegen

```

1 Initialisierung des Clusters;
2 foreach Szene  $s_i$  in  $S$  do
3   | Gruppierung von  $s_i$  gemäß dem Aufnahmezeitpunkt;
4 end
5 foreach Gruppe  $g$  in  $Gruppen_{\text{Aufnahmezeitpunkt}}$  do
6   | foreach Szene  $s_i$  in  $g$  do
7     | Kachel  $Tile_i =$  Schnittmenge des Zielbereichs und des von  $s_i$  abgebildeten
8     |   geographischen Bereichs;
9     | Zerschneidung von  $Tile_i$  in  $m_x * m_y$  quadratische Teilkacheln  $Teilkachel_{i_{m_x, m_y}}$ ;
10    end
11  foreach  $Teilkachel_{i_{m_x, m_y}}$  do
12    | Gruppierung von  $Teilkachel_{i_{m_x, m_y}}$  gemäß dem Aufnahmedatum und dem Spektralband;
13  end
14 Sortierung der Gruppen nach Aufnahmedatum;
15 // Approximierung der fehlenden Vegetationsindexwerte
16 foreach Gruppe  $g$  in  $Gruppen_{\text{Aufnahmezeitpunkt}, \text{Band}}$  do
17   | Transformation der einzelnen Teilkacheln in Pixelzeitreihen Approximierung der
18   |   fehlenden Werte für jede Pixelzeitreihe Schätzung für zukünftige Werte jeder
19   |   Pixelzeitreihe
20 end

```

Algorithm 1: Allgemeiner Algorithmus zur Approximierung von Pixelzeitreihen

Die Support Vector Regression ist ein Regressionsverfahren, das auf *Support Vector Machines*, auch Stützvektormaschinen genannt, aufbaut [DBK⁺97]. Support Vector Machines bezeichnen ein mathematisches Verfahren zur Mustererkennung, das seine Ursprünge im maschinellen Lernen hat. Dabei wird für einen Trainingsdatensatz der Form $\{(x_1, y_1), \dots, (x_n, y_n)\} \subset \chi \times \mathbb{R}$, wobei χ die räumliche Dimension der Eingabedaten angibt, eine Funktion $f(x)$ gesucht. Für $f(x)$ muss gelten, dass die Abweichungen $w_i < \epsilon \forall \{i | 1 \leq i \leq n\}$ sind. Es werden also nur Beobachtungswerte in das Modell mit einbezogen, die den durch ϵ definierten Abstand zwischen y_i und x_i nicht überschreiten. Darüber hinaus muss $\min_{w_i \leq \epsilon} \sum_{i=1}^n w_i$ gelten, um eine optimale Approximierung durch $f(x)$ zu erreichen.

Die Methode der kleinsten Quadrate ist ein Standardverfahren in der Mathematik, das Anfang des 19. Jahrhunderts von Carl Friedrich Gauß sowie Adrien-Marie Legendre unabhängig voneinander entwickelt wurde. Das Ziel der Methode ist es für eine Menge an Datenpunkten eine Funktion zu finden, die diese maximal annähert. Diese Funktion ermöglicht es für jede exogene Variable die entsprechende endogene Variable zu approximieren. Um die Funktion maximal anzunähern, wird die Summe der quadrierten Abstände der beobachteten Werte der endogenen Variablen von der Kurve minimiert. Im Gegensatz zu anderen Schätzmethoden ist es nicht notwendig die Verteilung der Ausgangsdaten anzunehmen. Dies ermöglicht die Anwendung der Methode der kleinsten Quadrate auf viele verschiedene Problemstellungen.

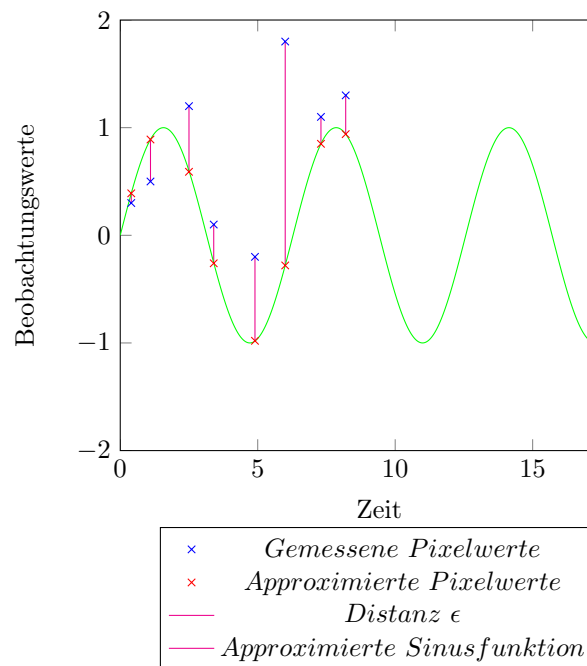


Abbildung 3.3: Graphische Darstellung der Berechnung der Distanz ϵ zwischen gemessenen und approximierten Werten

Um die Methode der kleinsten Quadrate sinnvoll anwenden zu können müssen die Ursprungsdaten bestimmte Voraussetzungen erfüllen. Die abhängige Größe y muss dabei von einer oder mehreren Größen, in unserem Fall x , abhängig sein. Diese Abhängigkeit wird durch die Funktion $y(x) = f(x; \alpha_1, \dots, \alpha_m)$ ausgedrückt, wobei f von x sowie m Funktionsparametern α_j abhängt. Die Abhängigkeitsfunktion muss dabei individuell gewählt oder mithilfe eines Modellierungsverfahrens bestimmt werden. Aufgrund der Verteilung der beobachteten Werte für den Vegetationsindex wird in diesem Fall eine trigonometrische Funktion als Ausgangsfunktion verwendet. Die Werte x_0, x_1, \dots, x_n entsprechen den Beobachtungszeitpunkten der ermittelten Werte y_0, y_1, \dots, y_n für den Vegetationsindex. Für alle y_0, y_1, \dots, y_n werden nun entsprechende α_j ermittelt, die eine möglichst gute Approximierung der beobachteten Daten durch die Funktion f ermöglichen. Des weiteren muss $n > m$ gelten, so dass mehr Beobachtungen als Funktionsparameter existieren.

Bei der Anwendung der Methode der kleinsten Quadrate wird versucht die Summe der quadrierten Abstände der beobachteten Werte y_0, y_1, \dots, y_n zu den durch f bestimmten Werten $\hat{y}_0, \hat{y}_1, \dots, \hat{y}_n$ zu minimieren. Die quadrierten Abstände werden Residuen genannt. Die Minimierung ist ein Optimierungsproblem, dass abhängig von der gewählten Abhängigkeitsfunktion gelöst werden muss. Dabei wird in lineare und nicht-lineare Funktionen unterschieden. Da wir eine trigonometrische Funktion nutzen, muss die Minimierung für nicht-lineare Abhängigkeitsfunktionen angewandt werden. Diese erfolgt mithilfe des Levenberg-Marquadt-Algorithmus, der das Gauß-Newton-Verfahren zur Lösung nicht-linearer Optimierungsprobleme mit einer Regularisierungstechnik kombiniert [Mor78]. Vllt. auch nicht, ist x in $\sin(x)$ linear?

Beschreibung Kombination der beiden Methoden + Überleitung zu Umsetzung

3.2 Umsetzung des Algorithmus mit Apache Flink

Im Rahmen dieser Arbeit wird der in Kapitel 3.1 beschriebene Algorithmus in Apache Flink implementiert. Dazu werden die in Kapitel 2.2.3 erwähnten Programmierschnittstellen für Java sowie Python genutzt. Grund für die Auswahl von Java sind die Stabilität und der Umfang der Java-Programmierschnittstelle von Flink. Die Motivation Python zu verwenden liegt darin begründet, dass viele Forscher Python zur Analyse von Daten nutzen und für die Analyse von Daten effizient implementierte Bibliotheken existieren, die viele wichtige Funktionen beinhalten. Diese Bibliotheken können auch bei der Nutzung von Python in Verbindung mit Flink weiterhin verwendet werden.

Grundlage für beide Implementation ist der vom Geographischen Institut der Humboldt-Universität zu Berlin umgesetzte Algorithmus, der in Python implementiert ist. Aufgrund der mangelnden Skalierbarkeit von Python wird eine verteilt agierende Variante des Algorithmus unter Nutzung von Flink implementiert. Die Umsetzung in Flink ist die mangelnde Skalierbarkeit von Python. Ziel der Arbeit ist der Vergleich der Laufzeit der verschiedenen Implementationen und die Beantwortung der Fragestellung, ob die Nutzung von Apache Flink zu einer Laufzeitverbesserung im Gegensatz zur bereits existierenden Implementation führt.

Wie bereits in Kapitel 2.2.3 beschrieben sind Flink-Programme Datenströme, die aus Datenquellen, Datentransformationen und Datenausgaben bestehen. Insofern kann nicht einfach das bestehende Programm übernommen werden sondern muss neu konzipiert werden. Der Datenfluss, den das Flink-Programm beschreibt, ist in Abbildung 3.4 abgebildet. In Schritt 1 des Datenflusses stellt eine Datenquelle die zu analysierenden Satellitenbilder im .bsq Format mit zugehörigen .hdr Dateien, die Metadaten bezüglich der Szene beinhalten, zur Verfügung. Da diese im Normalfall aus kompletten Szenen bestehen, müssen die Daten vor der Anwendung des in Kapitel 3.1 beschriebenen Algorithmus in kleinere rechteckige Teile, sogenannte Kacheln, zerteilt werden. Darüber hinaus können die Szenen möglicherweise auch Daten zu geographischen Gebieten beinhalten, die nicht Teil des zu analysierenden Gebiets sind. Um die Daten für die spätere Analyse aufzubereiten werden die Szenen in einem ersten Verarbeitungsschritt gemäß ihrem jeweiligen Aufnahmedatum gruppiert **Das kann ich vermutlich überspringen!**. Diese Gruppierung erfolgt durch die Anwendung der *GroupBy*-Transformation, die alle Kacheln anhand des Schlüsselwertes "Aufnahmezeitpunkt" transformiert und in Gruppen zusammenfasst. Aus allen so gruppierten Szenen werden dann mithilfe einer *groupReduce*-Funktion Kacheln zurecht geschnitten, die nur noch die für den durch den Nutzer bestimmten geographischen Bereich relevanten Teilszenen beinhalten. Nachdem die Ausgangsszenen auf den relevanten geographischen Bereich reduziert wurden, werden die Kacheln mithilfe des *Flatmap*-Operators erneut in kleinere Teilkacheln zerschnitten. Dies geschieht, um die Teilkacheln unabhängig voneinander pixelweise analysieren zu können. Die Kantenlängen der ungeschnittenen Kacheln müssen dabei ein ganzzahliges Vielfaches der Kantenlängen der Teilkacheln betragen, damit alle Pixel in genau einer Teilkachel enthalten sind. Dies bedeutet im Umkehrschluss, dass die Kantenlängen der ungeschnittenen Kacheln mindestens einen ganzzahligen Teiler besitzen müssen. Nachdem die Teilkacheln korrekt erzeugt wurden, werden sie wiederum gemäß ihrem "Aufnahmedatum" gruppiert. Anschließend wird der *Sort*-Operator genutzt, um die einzelnen Gruppen nach dem "Aufnahmedatum" zu sortieren. Dies ist notwendig, um die zeitliche Abfolge der Beobachtungen sicherzustellen, die bei der Analyse der Daten vorausgesetzt wird. Nachdem die Kacheln auf diese Weise für die Approximierung fehlender Werte vorbereitet wurden, kann die Analyse durchgeführt werden. Im Rahmen der Analyse werden die Approximierung fehlender sowie die Ermittlung der Erwartungswerte zukünftiger Ausprägungen des Vegetationsindex durchgeführt. Dies geschieht für jede Gruppe von Teilkacheln einzeln, so dass die Berechnungen auf verschiedenen TaskManagern durchgeführt werden können. Dazu werden sowohl bei der Nutzung der Java- als auch bei der Nutzung der Python-Programmierschnittstelle von Flink externe Bibliotheken genutzt, die in Abschnitt 3.1 beschriebenen Verfahren Support Vector Regression und

Methode der kleinsten Quadrate implementieren. Die Unterschiede bei der Nutzung der jeweiligen Bibliotheken werden in Sektion 3.2.2 aufgezeigt. Um die Verfahren anwenden zu können, werden zunächst aus jeder Gruppe von Teilkacheln derselben geographischen Position die Vegetationsindexwerte aller Pixel anhand ihrer geographischen Position in Pixelzeitreihen eingefügt. Folglich bilden alle Vegetationsindexwerte, deren zugehörige Pixel dieselbe Koordinate teilen, eine Liste, die nach dem jeweiligen Aufnahmezeitpunkt sortiert ist. Jede Pixelzeitreihe wird nun als Trainingsdatensatz für die Support Vector Regression genutzt. **Werden die nicht aufgeteilt ca. 70/30 in Training und Testdaten?**. Anhand der Trainingsdaten wird dann eine gemäß der Definition in Kapitel 3.1 optimale Approximierungsfunktion f berechnet.

3.2.1 Komplexitätsanalyse

Kurze Einleitung, Beschreibung meiner Kompl.A. (Big O Notation kurz erläutern)
Kompl.A. für den Algorithmus

3.2.2 Unterschiede zwischen der Nutzung der Java-Programmierschnittstelle und der Python-Programmierschnittstelle

Die Implementation des Algorithmus als Apache Flink-Programm erfolgt über die Nutzung der jeweiligen Programmierschnittstellen, die für Java und Python zur Verfügung stehen. Aufgrund der Verschiedenheit von Java und Python sind die Implementationen nicht identisch, auch wenn sie identische Ergebnisse produzieren. Dies liegt zum einen daran, dass die Spracheigenschaften von Python, die in Kapitel 2.2.4 beschrieben wurden, und Java unterschiedlich sind. Zum anderen unterscheidet sich auch der Funktionsumfang der beiden Schnittstellen. **Ist das so?**. Diese Unterschiede müssen bei der Implementation berücksichtigt werden.

Wie bereits in Sektion 2.2.4 erwähnt, verfügt Python über die Möglichkeit Teile der Implementation in C statt in Python zu implementieren. Diese werden vom Python-Interpreter erkannt und direkt in C ausgeführt, was die Verarbeitungsgeschwindigkeit dieses Teils des Algorithmus steigern kann. Dementsprechend sind einige Teile der für die Regression genutzten Python-Bibliotheken NumPy und SciPy in C geschrieben und lediglich in Python eingebettet. Diese werden dann von C

Java hingegen profitiert von der umfangreicheren Flink-Schnittstelle, die mehr Operatoren umfasst als die Python-Schnittstelle. Insbesondere das Fehlen des *SortBy*-Operators **Fehlt der wirklich?** erfordert eine leichte Veränderung des Datenstroms 3.4. **Veränderung beschreiben**

3.3 Umsetzung des Algorithmus in Python

Zusätzlich zu den Implementationen des Algorithmus in Apache Flink mittels der Java- bzw. der Python-Programmierschnittstelle wird eine weitere Implementierung in Python betrachtet. Diese wurde von Mitarbeitern des Geographischen Instituts der Humboldt-Universität zu Berlin umgesetzt und liegt dem Vergleich der drei Implementierungsvarianten zugrunde. Die Implementation besitzt dabei die in Sektion 2.2.4 beschriebenen Vor- und Nachteile eines Python-Programms. Zwar lässt sich der Analysealgorithmus mithilfe von effizient umgesetzten Bibliotheken so umsetzen, dass die Berechnungszeit auf sequentiell arbeitenden Maschinen gut ist. Jedoch lässt sich das Programm nicht automatisiert parallelisieren, so dass der Algorithmus bei zunehmenden Datenmengen nicht skalieren lässt. Aufgrund der Charakteristika der Problemstellung, die eine Aggregation von Satellitenbildern fordert, wobei die Anzahl der Szenen sich fortlaufend erhöht, ist dies insbesondere in unserem Fall problematisch.

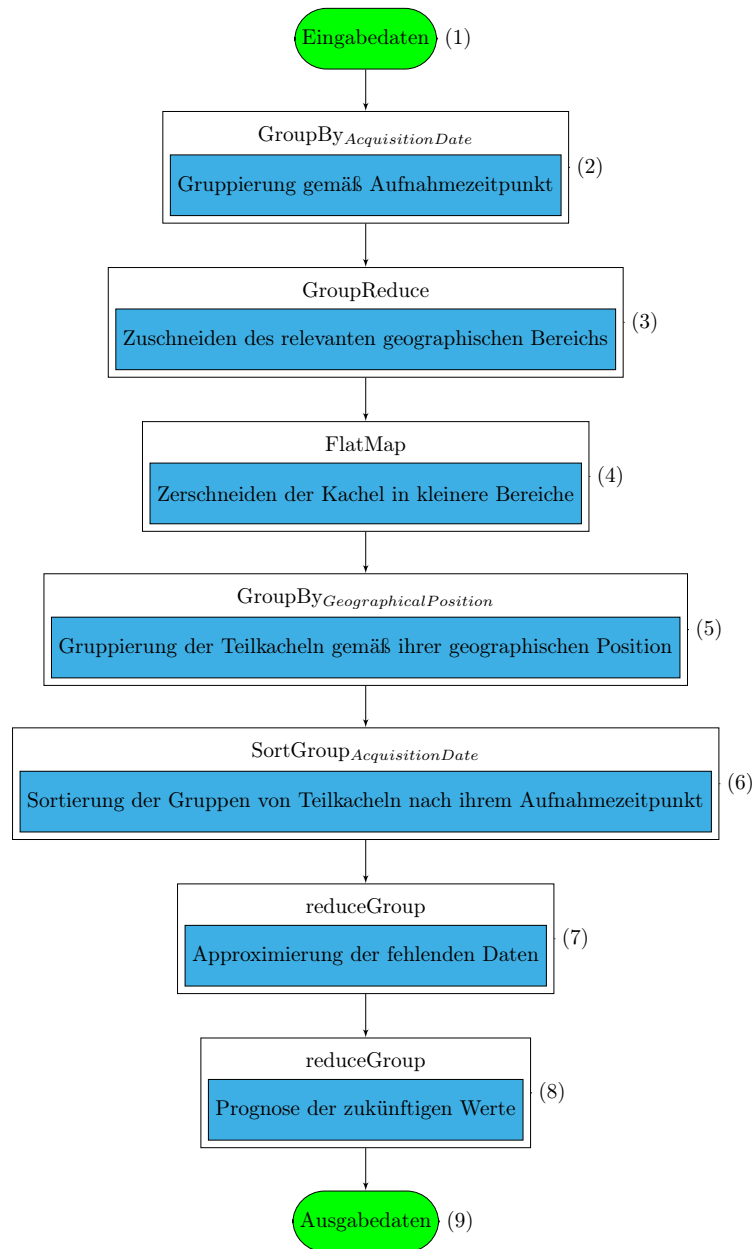


Abbildung 3.4: Visualisierung des Datenstroms des Analysealgorithmus

Kapitel 4

Evaluierung

Ziel der Untersuchungen in dieser Bachelorarbeit ist die Evaluierung der parallelisierten Implementationen des in Sektion 3.1 beschriebenen Algorithmus zur Analyse von Veränderungen des Vegetationsindex. Dabei sollen nicht nur die beiden Varianten des Algorithmus, die die Flink-Programmierschnittstellen für Java und Python nutzen, untereinander verglichen werden sondern auch eine bereits existierende Variante, die nativ in Python implementiert ist, in die Untersuchung mit einbezogen werden. Zum Testen der verteilten Algorithmen wird ein Cluster von Maschinen verwendet, während für die native Pythonanwendung eine einzelne Maschine verwendet wird, da sie nicht parallelisierbar ist. Als Testdaten werden Satellitendaten verwendet, die vom Landsat-System über mehrere Jahre hinweg aufgenommen wurden. Die Untersuchung soll zeigen, wie sich das Laufzeitverhalten der einzelnen Varianten ändert, wenn die Eingabedatenmenge vergrößert oder andere Parameter wie zum Beispiel die Anzahl der Teilkacheln verändert werden.

4.1 Evaluierungskriterien

Um die drei verschiedenen Implementationen des Algorithmus vergleichen zu können, müssen zunächst Vergleichsmetriken und -kriterien definiert werden. Als Hauptkriterien werden die zwei in Sektion 2.2.5 beschriebenen Kriterien Skalierbarkeit und Ausführungsgeschwindigkeit genutzt. Das Verhältnis des Preises zur Leistung der genutzten Hardware wird dabei außen vorgelassen, da der Fokus der Evaluierung auf der Skalierbarkeit und der Ausführungsgeschwindigkeit liegt.

Insbesondere die Skalierbarkeit der genutzten Implementation eines Algorithmus zur Verarbeitung von Satellitenbildern ist von großer Bedeutung, da die zu verarbeitende Datenmenge mit jeder zu betrachtenden Szene um etwa 1500 Megabyte ansteigt (bei einer Nutzung von Bildmaterial, das von Landsat 8 aufgenommen wurde). Da bei einer durchschnittlichen Überflugfrequenz einer geographischen Region von 16 Tagen pro Jahr circa 22 Szenen von jeder dieser Regionen angefertigt werden, steigt die Datenmenge für jedes Jahr, das in die Analyse mit einbezogen werden soll, um circa 36 Gigabyte. Um eine Analyse des Vegetationsindex einer Region tätigen zu können, werden die Daten mehrerer Jahre verwendet, so dass die Eingabedatenmenge für den Algorithmus leicht mehr als 100 Gigabyte für jede Region betragen kann. Soll die Analyse auch ältere Daten des seit 1972 aktiven Landsat-Systems nutzen, vervielfacht sich die Eingabedatenmenge.

Die Ausführungsgeschwindigkeit ist wie in Sektion 2.2.5 beschrieben als das Verhältnis von geleisteter Berechnungsarbeit pro Zeiteinheit definiert. Daraus folgt, dass sich die Ausführungsgeschwindigkeit des Algorithmus verlängert, wenn die Datenmenge ansteigt und gleichzeitig die sonstigen Parameter, wie beispielsweise die genutzte Hardware, unverändert bleiben. Zu lange Laufzeiten des Algorithmus sind jedoch bei der Nutzung zur Analyse von vielen geographischen

Regionen von Nachteil. Um lange Laufzeiten zu vermeiden, muss also entweder der Algorithmus optimiert, die genutzte Rechenkapazität erhöht oder die Eingabedatenmenge verkleinert beziehungsweise klein gehalten werden. Im Rahmen der durchgeführten Tests wird lediglich die letzte dieser Möglichkeiten betrachtet. Aufgrund der vorgegebenen zur Verfügung gestellten Hardware wird eine Variation dieser nicht berücksichtigt. Eine optimierte Version des Algorithmus ist ein möglicher weiterer Schritt, der auf den in dieser Bachelorarbeit erarbeiteten Resultaten aufbaut.

4.2 Versuchsbeschreibung

Um die drei Implementationsvarianten des Algorithmus anhand der in Sektion 4.1 erläuterten Evaluationskriterien beurteilen zu können, müssen alle Varianten mithilfe identischer Testdatensätzen ausgeführt werden. Dabei werden alle Varianten mit mehreren Testdatensätzen ausgeführt, um den Einfluss der Größe der Eingabedaten auf die Kriterien zu evaluieren. Dabei sind insbesondere anwendungstypische Datenmengen zu verwenden, die die Szenen von einem bis zu mehreren Jahren umfassen können. Die Vergrößerung der Eingabedatenmenge ermöglicht auch die Bestimmung der Skalierbarkeit der jeweiligen Implementationsvariante. Zusätzlich wird die Anzahl der Teilkacheln verändert.

Im Rahmen der Tests, die zur Evaluierung der Skalierbarkeit und der Ausführungsgeschwindigkeit durchgeführt werden, wird jede Implementationsvariante 3 verschiedene Eingabedatenmengen unterschiedlicher Größe verarbeiten. Zusätzlich wird die Anzahl der Teilkacheln variiert, um zu ermitteln, ob dies einen relevanten Einfluss auf die Ausführungsgeschwindigkeit hat. Die in Tabelle 4.1 beschriebenen Testkonfigurationen werden einen Umfang von 22 Szenen, 66 Szenen beziehungsweise 220 Szenen haben. Dies entspricht einem Zeitraum von circa 1, 3 und 10 Jahren, so dass sowohl Analysen über kurze als auch über lange Zeiträume untersucht werden. Das ist notwendig, um die Skalierbarkeit beziehungsweise die Auswirkung der Größe der Eingabedatenmenge auf die Ausführungsgeschwindigkeit zu untersuchen.

Die Größe des Zielbereichs wird für alle Versuche fest definiert, so dass die Ergebnisse vergleichbar sind. Bei der Anzahl der Teilkacheln werden hingegen mehrere Werte untersucht, wobei dieser im Intervall $1 \leq \#Teilkacheln_{Szenen} \leq \#Pixels_{Szenen}$ liegen muss. Auf diese Weise soll die optimale Anzahl an Teilkacheln für die verschiedenen Implementierungen ermittelt werden. Dies spielt vermutlich insbesondere bei den parallelisiert implementierten Flink-Varianten des Algorithmus eine wichtige Rolle, da die Anzahl der Teilkacheln an einigen Stellen des Algorithmus den Grad der Parallelisierbarkeit bestimmt. Bei der dritten Variante, der auf einer einzelnen Maschine ausgeführten Python-Implementation, dürfte dieser Parameter keine Auswirkung haben, da keine nebenläufige Berechnungen genutzt werden. Bei allen Versuchen gilt, dass eine möglichst große Vergleichbarkeit der Tests und somit auch der Testergebnisse gewährleistet werden muss. Nur so können die Ergebnisse der Tests für die Skalierbarkeit und die Ausführungsgeschwindigkeit jeder Variante verglichen werden. Insgesamt ergeben sich also sechs Testkonfigurationen, die durch jede der drei Implementationsvarianten verarbeitet werden. Dabei wird jeder Test mehrfach **Später genau spezifizieren** durchgeführt und als endgültiges Resultat ein Durchschnittswert der Messungen verwendet, so dass der Einfluss von Messfehlern und zufälligen Störfaktoren auf die Ergebnisse verringert werden kann.

Um zu ermitteln, ob die Faktoren “Größe der Eingabedatenmenge” und “Anzahl der Teilkacheln” unabhängig voneinander sind, wird zusätzlich noch eine dritte Versuchsreihe durchgeführt. Bei dieser werden beide Parameter gleichzeitig verändert, so dass eine mögliche Abhängigkeit nachgewiesen werden kann. **Eventuell lasse ich das weg. Hab ja jetzt schon x Durchläufe von insgesamt 18 Tests.**

Beschreibung der Hardware

Versuch	Anzahl der Szenen	Eingabedatenmenge in GB	Größe des Zielbereichs in Pixeln	Größe der Teilkacheln in Pixel
Versuch #1	22	xxx	xxx1	xxx1
Versuch #2	22	xxx	xxx1	xxx2
Versuch #3	66	xxx	xxx1	xxx1
Versuch #4	66	xxx	xxx1	xxx2
Versuch #5	220	xxx	xxx1	xxx1
Versuch #6	220	xxx	xxx1	xxx2

Tabelle 4.1: Beschreibung der für die Evaluation genutzten Testkonfigurationen

4.3 Auswertung

Beschreibung und Bewertung der Ergebnisse meiner Untersuchungen (inklusive Charts, Begründung und Erwartungen). Nutzung der Python-Impl. als Vergleichsimplementation

Kapitel 5

Fazit

Fazit und Ausblick z.b. Vergleich mit anderen Untersuchungen

Literaturverzeichnis

- [AAP⁺15] Shadab Alam, Franco D. Albareti, Carlos Allende Prieto, F. Anders, Scott F. Anderson, Timothy Anderton, Brett H. Andrews, Eric Armengaud, Áric Aubourg, Stephen Bailey, and et al. The eleventh AND twelfth data releases of the sloan digital sky survey: Final data from sdss-iii. *The Astrophysical Journal Supplement Series*, 219(1):12, Jul 2015.
- [ABE⁺14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, and et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939,964, May 2014.
- [AKL⁺12] Sattam Alsubaiee, Young-Seok Kim, Chen Li, Nicola Onose, Pouria Pirzadeh, Rares Vernica, Jian Wen, Yasser Altowim, Hotham Altwaijry, Alexander Behm, and et al. Asterix. *Proceedings of the VLDB Endowment*, 5(12):1898,1901, Aug 2012.
- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS ,Ãd67 (Spring)*, 1967.
- [Ast] Asterixdb website. <https://asterixdb.ics.uci.edu/>.
- [Bak10] Monya Baker. Next-generation sequencing: adjusting to data overload. *Nat Meth*, 7(7):495,Ãi499, Jul 2010.
- [BBN⁺13] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. Setting the direction for big data benchmark standards. *Lecture Notes in Computer Science*, page 197,Ãi208, 2013.
- [Bea00] David M Beazley. Scientific computing with python. In *Astronomical Data Analysis Software and Systems IX*, volume 216, page 49, 2000.
- [BEH⁺10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephelē/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.
- [Bel71] C Gordon Bell. C. mmp: the cmu multiminiprocessor computer, requirements and overview of the initial design. 1971.
- [BH85] A. Borodin and J.E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130,Ãi145, Feb 1985.

- [Bor13] Dhruba Borthakur. Petabyte scale databases and storage systems at facebook. *Proceedings of the 2013 international conference on Management of data - SIGMOD*, 2013.
- [BPP07] Debasish Basak, Srimanta Pal, and Dipak Chandra Patranabis. Support vector regression. *Neural Information Processing-Letters and Reviews*, 11(10):203–224, 2007.
- [BR09] Stefan Bethge and Astrid Rheinländer. Mapreduce. Seminararbeit, Humboldt-Universität zu Berlin, 2009.
- [Bra] Mikio Braun. The future of big data (according to stratosphere/flink). <http://blog.mikiobraun.de/2014/06/future-big-data-flink-stratosphere.html>.
- [Cha11] Anju Chaudhary. Thermal infrared sensors. *Encyclopedia of Snow, Ice and Glaciers*, page 1156, 2011.
- [CRK14] Yanpei Chen, Francois Raab, and Randy Katz. From tpc-c to big data benchmarks: A functional workload model. *Lecture Notes in Computer Science*, page 28, 2014.
- [DBK⁺97] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 9*, pages 155–161. MIT Press, 1997.
- [Deu12] Deutsches Institut für Normung e.V. *Photogrammetrie und Fernerkundung - Begriffe*, 8 2012. Rev. 3.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. page 13, 2004.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce. *Communications of the ACM*, 51(1):107, Jan 2008.
- [EMC14] EMC². The digital universe of opportunities. Technical report, EMC², 2014.
- [Foua] Apache Software Foundation. Apache flink website. https://ci.apache.org/projects/flink/flink-docs-release-0.9/internals/job_scheduling.html.
- [Foub] Apache Software Foundation. Apache spark website. <https://spark.apache.org/>.
- [Fouc] Apache Software Foundation. Apache storm website. <https://storm.apache.org/>.
- [Foud] Apache Software Foundation. Flink website. <https://flink.apache.org/>.
- [Foue] Apache Software Foundation. Hadoop website. <https://hadoop.apache.org/>.
- [Fou15] Apache Software Foundation. The apache software foundation announces apacheTM flinkTM as a top-level project. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69, January 2015.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29, Dec 2003.
- [GP] GfZ-Potsdam. Geomultisens website. <http://www.geomultisens.gfz-potsdam.de/>.

- [Gra92] Jim Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
- [HDF13] Kai Hwang, Jack Dongarra, and Geoffrey C Fox. *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan Kaufmann, 2013.
- [HPS⁺12] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, 5(11):1256,1267, Jul 2012.
- [IDB12] James R. Irons, John L. Dwyer, and Julia A. Barsi. The next landsat satellite: The landsat data continuity mission. *Remote Sensing of Environment*, 122:11,21, Jul 2012.
- [Jac09] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36, August 2009.
- [Jon13] M Tim Jones. Process real-time big data with twitter storm. *IBM Technical Library*, 2013.
- [Lan01] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *Application Delivery Strategies published by META Group Inc.*, Feb 2001.
- [MA11] K. Jarrod Millman and Michael Aivazis. Python for scientists and engineers. *Computing in Science & Engineering*, 13(2):9,Ä12, Mar 2011.
- [Mar06] Alex Martelli. *Python in a Nutshell. A Desktop Quick Reference*. O'Reilly, second edition edition, 2006.
- [MHW⁺08] Jeffrey G. Masek, Chengquan Huang, Robert Wolfe, Warren Cohen, Forrest Hall, Jonathan Kutler, and Peder Nelson. North American forest disturbance mapped from a decadal landsat record. *Remote Sensing of Environment*, 112(6):2914,Ä2926, Jun 2008.
- [Mor78] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.
- [MSWI04] B.L. Markham, J.C. Storey, D.L. Williams, and J.R. Irons. Landsat sensor performance: history and current status. *IEEE Transactions on Geoscience and Remote Sensing*, 42(12):2691,2694, Dec 2004.
- [Oli07] Travis E Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.
- [Pad97] Fabian Padge. Fusion von landsat-tm und spot-hrv daten zur ableitung einer satelitenbildkarte von rostock und die exemplarische nutzung der daten zur flächennutzungsklassifizierung. Master's thesis, Universität Hamburg, 1997.
- [Pem15] Giridhar Pemmasani. Dispy website. <http://dispy.sourceforge.net/>, 11 2015.
- [Sei85] Charles L Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.
- [TW12] Arthur Trew and Greg Wilson. *Past, present, parallel: a survey of available parallel computer systems*. Springer Science & Business Media, 2012.

- [Van15] Vitalii Vanovschi. Parallel python website. <http://www.parallelpython.com/>, 11 2015.
- [WB72] William A Wulf and C Gordon Bell. C. mmp: a multi-mini-processor. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, pages 765–777. ACM, 1972.
- [YAJEA⁺00] Donald G. York, J. Adelman, Jr. John E. Anderson, Scott F. Anderson, James Annis, Neta A. Bahcall, J. A. Bakken, Robert Barkhouser, Steven Bastian, Eileen Berman, William N. Boroski, Steve Bracker, Charlie Briegel, John W. Briggs, J. Brinkmann, Robert Brunner, Scott Burles, Larry Carey, Michael A. Carr, Francisco J. Castander, Bing Chen, Patrick L. Colestock, A. J. Connolly, J. H. Crocker, Istvan Csabai, Paul C. Czarapata, John Eric Davis, Mamoru Doi, Tom Dombeck, Daniel Eisenstein, Nancy Ellman, Brian R. Elms, Michael L. Evans, Xiaohui Fan, Glenn R. Federwitz, Larry Fiscelli, Scott Friedman, Joshua A. Frieman, Masataka Fukugita, Bruce Gillespie, James E. Gunn, Vijay K. Gurbani, Ernst de Haas, Merle Haldeman, Frederick H. Harris, J. Hayes, Timothy M. Heckman, G. S. Hennessy, Robert B. Hindsley, Scott Holm, Donald J. Holmgren, Chi hao Huang, Charles Hull, Don Husby, Shin-Ichi Ichikawa, Takashi Ichikawa, Zeljko Ivezifá, Stephen Kent, Rita S. J. Kim, E. Kinney, Mark Klaene, A. N. Kleinman, S. Kleinman, G. R. Knapp, John Korienek, Richard G. Kron, Peter Z. Kunszt, D. Q. Lamb, B. Lee, R. French Leger, Siriluk Limmongkol, Carl Lindenmeyer, Daniel C. Long, Craig Loomis, Jon Loveday, Rich Lucinio, Robert H. Lupton, Bryan MacKinnon, Edward J. Mannery, P. M. Mantsch, Bruce Margon, Peregrine McGehee, Timothy A. McKay, Avery Meiksin, Aronne Merelli, David G. Monet, Jeffrey A. Munn, Vijay K. Narayanan, Thomas Nash, Eric Neilsen, Rich Neswold, Heidi Jo Newberg, R. C. Nichol, Tom Nicinski, Mario Nonino, Norio Okada, Sadanori Okamura, Jeremiah P. Ostriker, Russell Owen, A. George Pauls, John Peoples, R. L. Peterson, Donald Petravick, Jeffrey R. Pier, Adrian Pope, Ruth Pordes, Angela Prosapio, Ron Rechenmacher, Thomas R. Quinn, Gordon T. Richards, Michael W. Richmond, Claudio H. Rivetta, Constance M. Rockosi, Kurt Ruthmansdorfer, Dale Sandford, David J. Schlegel, Donald P. Schneider, Maki Sekiguchi, Gary Sergey, Kazuhiro Shimasaku, Walter A. Siegmund, Stephen Smee, J. Allyn Smith, S. Snedden, R. Stone, Chris Stoughton, Michael A. Strauss, Christopher Stubbs, Mark SubbaRao, Alexander S. Szalay, Istvan Szapudi, Gyula P. Szokoly, Anirudda R. Thakar, Christy Tremonti, Douglas L. Tucker, Alan Uomoto, Dan Vanden Berk, Michael S. Vogeley, Patrick Waddell, Shu i Wang, Masaru Watanabe, David H. Weinberg, Brian Yanny, and Naoki Yasuda. The sloan digital sky survey: Technical summary. *The Astronomical Journal*, 120(3):1579, 2000.
- [ZCF⁺10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.
- [ZdP⁺12] Paul Zikopoulos, Dirk deRoos, Krishnan Parasuraman, Thomas Deutsch, James Giles, and David Corrigan. *Harness the Power of Big Data The IBM Big Data Platform*. McGraw-Hill Osborne Media, 2012.
- [ZWO12] Zhe Zhu, Curtis E Woodcock, and Pontus Olofsson. Continuous monitoring of forest disturbance using all available landsat imagery. *Remote Sensing of Environment*, 122:75–91, 2012.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine ...arbeit in diesem Studienggebiet erstmalig einzureichen.

Berlin, den 22. Dezember 2015

.....

Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, 22nd December 2015

.....