



# 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.  
Prof. Dr. Klaus Bothe

eingereicht am: .....

verteidigt am: .....



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>3</b>
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
<b>3</b>	<b>Algorithmus zur Analyse von Pixelzeitreihen</b>	<b>15</b>
3.1	Beschreibung des Algorithmus . . . . .	15
3.1.1	Support Vektor Regression . . . . .	18
3.1.2	Komplexitätsanalyse der Algorithmen . . . . .	19
3.2	Umsetzung des Algorithmus mit Apache Flink . . . . .	25
3.2.1	Unterschiede zwischen den drei Implementationsvarianten . . . . .	27
<b>4</b>	<b>Evaluierung</b>	<b>29</b>
4.1	Evaluierungskriterien . . . . .	29
4.2	Versuchsbeschreibung und -erwartungen . . . . .	30
4.3	Auswertung . . . . .	32
4.3.1	Auswertung der Evaluierung der parallelisierten Varianten . . . . .	32
4.3.2	Auswertung der Evaluierung der Python-Variante . . . . .	36
4.3.3	Vergleich der parallelisierten und der nicht-parallelisierten Varianten . . . . .	38
<b>5</b>	<b>Fazit</b>	<b>42</b>
<b>A</b>	<b>Testkonfigurationen der Evaluation des Algorithmus</b>	<b>44</b>
A.1	Konfigurationen der parallelisierten Varianten . . . . .	44
A.1.1	Konfiguration für variable $n_s$ . . . . .	44
A.1.2	Konfiguration für variable $dim_k$ . . . . .	44
A.2	Zweiter Abschnitt . . . . .	44

# 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<sup>+</sup>00, AAP<sup>+</sup>15]. Jede Nacht kommen circa 250 Gigabyte neu hinzu. 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 „Speicherklippe“ 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 eine ebenfalls zu realisierende Python-Implementation genutzt. Sie sollte die untere Schranke der Leistungsmessungen darstellen. Die zweite und dritte Variante werden in Flink implementiert. Diese beiden Varianten unterscheiden sich bezüglich der genutzten Programmiersprache. Schließlich werden alle drei Varianten unter möglichst identischen Bedingungen getestet. Dies bedeutet, dass sowohl die Testumgebung als auch die Testdaten identisch sein sollen. Dabei sollen alle Varianten wahlweise auf einer leistungsfähigen Einzelmaschine oder auf 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. Die Satellitenbilder, die in dieser Bachelorarbeit als Ausgangsdaten für die Untersuchungen genutzt werden, sind von Landsat 5 aufgenommen worden, der 1984 gestartet wurde und bis Ende 2012

im Einsatz war. Er besaß zwei Instrumente zur Fernerkundung, den *Thematic Mapper* und das *Multispectral Scanner System*.

Der Thematic Mapper ist ein optomechanisches Instrument, das emittierte elektromagnetische Strahlung im Spektralbereich von  $0,45\ \mu\text{m}$  bis  $2,35\ \mu\text{m}$  unterteilt in 6 Spektralkanäle sowie einen Infrarot-Kanal, der Licht mit Wellenlängen zwischen  $10,41\ \mu\text{m}$  und  $2,5\ \mu\text{m}$  erfasst. Mit Ausnahme des Infrarot-Kanals, der eine Auflösung von  $120 \times 120$  Metern pro Pixel besitzt, lösen alle Bänder mit  $30 \times 30$  Metern pro Pixel auf. Mit beiden Instrumenten deckt Landsat 5 damit die Bereiche des für das menschliche Auge sichtbaren Lichts sowie des nahen und des mittleren Infrarotlichts ab.

Ergänzt wurde der Thematic Mapper durch das Multispectral Scanner System. Ebenfalls ein optomechanisches System, das vier weitere Spektralbänder besaß, die von der Erde reflektierte Lichtemissionen im Wellenlängenbereich von  $0,5\ \mu\text{m}$  -  $1,1\ \mu\text{m}$  registrierten. Diese hatten eine Auflösung von circa  $68 \times 83$  Metern, die üblicherweise auf eine Auflösung circa  $60 \times 60$  Meter abgebildet wurden.

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  $600\text{--}700\ \mu\text{m}$  sowie des nahen Infrarotbereichs zwischen  $700$  und  $1300\ \mu\text{m}$ . 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] Daten mehrerer Spektralbänder 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<sup>+</sup>08].

Landsat 5 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 172 Kilometer für Bilder Thematic Mappers, das Multispectral Scanner System nimmt quadratische Szenen mit 185 Kilometer Kantenlänge auf. 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 5 überflogen, so dass jährlich mindestens circa 22 Aufnahmen eines geographischen Bereichs gemacht werden [IDB12].

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<sup>+</sup>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<sup>+</sup>00, AAP<sup>+</sup>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].

**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.

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 zangsweise sequentiell durchgeführt werden müssen. Jedes Programm besitzt solche Programmteile, etwa Speicherallokationen oder der sequentielle bzw. synchrone Zugriff von Threads auf geteilte Ressourcen, so dass Amdahls Gesetz für alle Programme gilt.

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.

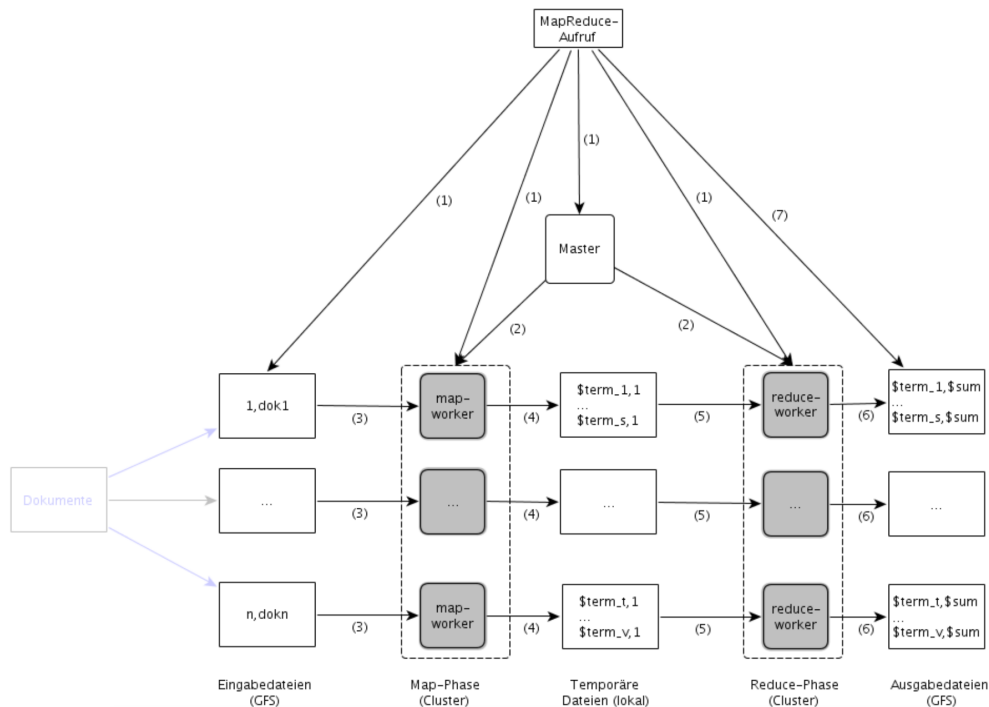


Abbildung 2.1: Systemübersicht eines Map-Reduce System [BR09]

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.

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 [Fou] und anderen Systemen wie beispielsweise Spark [ZCF<sup>+</sup>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<sup>+</sup>10], Apache Storm [Fouc, Jon13] oder Asterix [Ast, AKL<sup>+</sup>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<sup>+</sup>10, ABE<sup>+</sup>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. Dieser wird *Operator DAG* genannt.

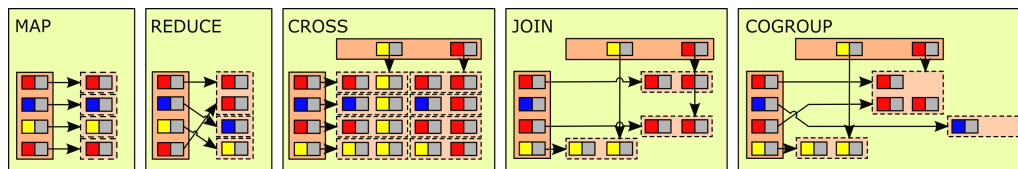


Abbildung 2.2: Funktionen zweiter Ordnung: (a) Map, (b) Reduce, (c) Cross, (d) Match und (e) CoGroup [HPS<sup>+</sup>12]

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	$R \times S \times K \times F \times f \rightarrow [f(r_1^{v_1}, \dots, r_{n_1}^{v_1}, s_1^{v_1}, \dots, r_{m_1}^{v_1}), \dots, f(r_1^{v_l}, \dots, r_{n_l}^{v_l}, s_1^{v_l}, \dots, r_{m_l}^{v_l})]$ 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<sup>+</sup>12]

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 parallelisierten ausführbaren Datenfluss, den *OptimizedPlan*. Dieser Datenfluss soll möglichst stark parallelisierbar sein, damit eine effiziente Verarbeitung durch die Flink-Laufzeitumgebung 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 initialisiert. 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

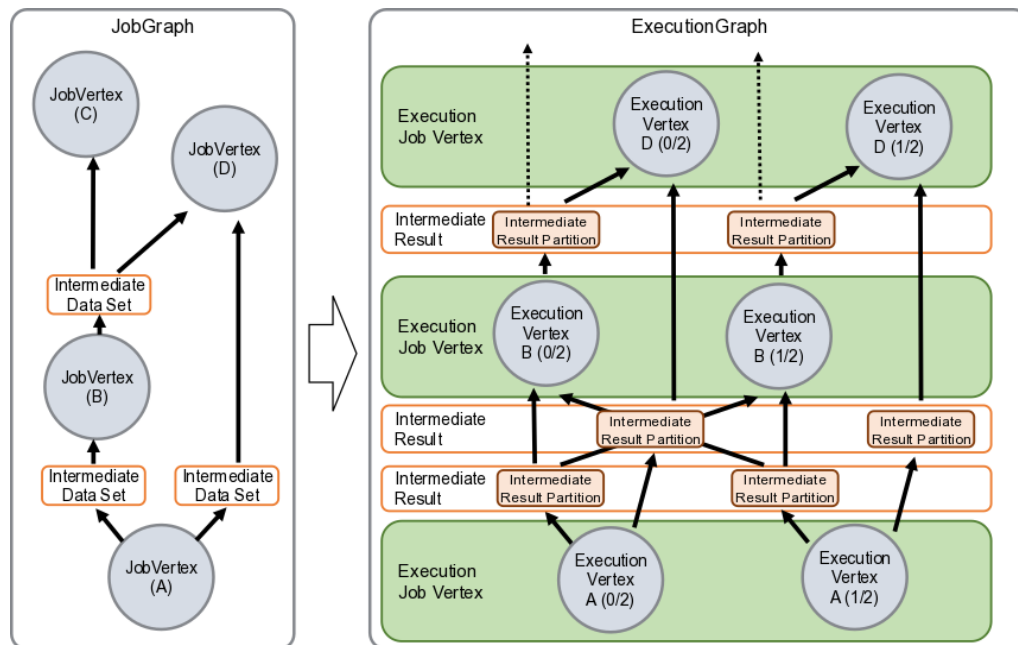


Abbildung 2.3: Modell eines Jobgraphs und des dazugehörigen ExecutionGraphs [Foua]

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 ausführbar 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 leistungintensive Programmteile direkt als C- oder C++-Code eingebunden und ausgeführt werden. Ein weiterer Vorteil von Python ist die starke Typisierung und daraus resultierend die bes-

sere 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<sup>+</sup>13].

Aufgrund des in Sektion 2.2.1 beschriebenen Notwendigkeiten für parallele Datenverarbeitung vergrößern sich auch die zu verarbeitenden Datenmengen kontinuierlich. Deshalb ist die **Skalier-**



**barkeit** des datenverarbeitenden Systems für Big-Data Anwendungen besonders wichtig. Ein Datenverarbeitungssystem gilt dann als skalierbar, wenn es flexibel auf die zu verarbeitende Datenmenge angepasst werden kann. Dies muss sowohl für die Software- als auch für die Hardwarekomponenten eines Systems gelten. Bei Clustern sind aufgrund ihrer parallelisierten Architektur, die aus mehreren miteinander verbundenen Maschinen besteht, weitere Maschinen teilweise sehr einfach integrierbar. Sie müssen häufig lediglich an die bestehende Netzwerkinfrastruktur des Systems angeschlossen werden. Auch die Systeme, die wie das Map-Reduce-System auf eine parallelisierte Ausführung spezialisiert sind, ermöglichen es einen Grad der Parallelität auszuwählen, so dass die erweiterten Kapazitäten eines Clusters genutzt werden können. Dabei ist wie bei jeder parallelisierten Architektur Amdahls Gesetz zu beachten, dass eine maximal linear mit der Anzahl der Prozessorkerne ansteigende Beschleunigung der Programmausführung möglich ist [Amd67]. Des weiteren steigt mit der Vergrößerung des Grades der Parallelität zwar die nutzbare Rechenleistung, gleichzeitig entstehen aber auch längere Kommunikationszeiten zwischen den einzelnen Maschinen des genutzten Clusters. Im Gegensatz zu parallelisierten Architekturen ist bei sequentiell arbeitenden Datenverarbeitungssystemen die Skalierbarkeit 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 Netzwerkknotten wie bereits beim Map-Reduce System beschrieben 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 abzuberechnen.

Die Skalierbarkeit hat auch unmittelbare Folgen für die weiteren Bewertungskriterien, insbesondere für die **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 dieses Indexes zu messen, werden die durch Landsat Satelliten aufgezeichneten Szenen der Zielregion dahingehend analysiert. Auf Basis dieser Analyse der in der Vergangenheit gemessenen Werte können dann zukünftige Indexwerte prognostiziert werden und etwaige Abweichungen des zukünftig tatsächlich gemessenen Vegetationsindex automatisiert ermittelt werden. Die zu diesem Zweck eingesetzte Analyse ist eine Anwendung der *Support Vector Regression* (SVR) [BPP07], die in Abschnitt 3.1.1 beschrieben wird. Die Ergebnisse der Analyse werden dann im Rahmen des *Continuous Monitoring of Forest Disturbance Algorithm* (CMFDA) [ZWO12] genutzt. Dieser wurde 2012 von Zhu et al. vorgestellt [ZWO12] und wurde entwickelt, um die Veränderung von Waldbeständen geographischer Regionen kontinuierlich zu überwachen und atypische Veränderungen zu entdecken.

In Tabelle 3.1 ist die Bedeutung der genutzten Notationen erläutert.

Der Algorithmus 1 beschreibt die Analyse der Veränderung des Vegetationsindex der Pixel einer geographischen Region. Dieser Algorithmus benötigt eine Menge von Landsat-Szenen  $S$  sowie die Koordinaten der Zielregion als Eingabe. Die genutzten Szenen beinhalten dabei möglicherweise Daten von mehreren Spektralbändern. Die Szenen werden in Zeile 6 zunächst gemäß ihres Aufnahmedatums  $t \in T$  gruppiert. Mithilfe der Schleife in Zeile 11 wird aus jeder Szene  $s_t \in S$  die exakte Zielregion  $G$  ausgeschnitten. Dieser Schritt ist notwendig, da etwaige Zielregionen zumeist kleiner sind als eine ursprüngliche Landsat-Szene oder nur ein Teil der Zielregion tatsächlich in der Ausgangsszene abgebildet ist. Der ausgeschnittene Bereich wird als Kachel  $k_{t,b}$  bezeichnet und wird durch ihr Spektralband  $b \in B$  und den Aufnahmezeitpunkt  $t$  charakterisiert. In Zeile 12 werden anschließend alle Kacheln anhand ihrer Bänder gruppiert, so dass für alle Kacheln  $k_{t,b} \in K_b$  gilt. Um die Eigenschaften und Ordnung der Kacheln zu verdeutlichen, wird ein Würfel mit den Dimensionen Breite der Kachel, Höhe der Kachel und dem Aufnahmezeitpunkt der Kacheln modelliert.

Notation	Bedeutung
$B$	Menge der relevanten Spektralbänder $b$
$cv_{pos\_px,b}$	Koeffizientenvektor der Pixelzeitreihe $pts_{pos\_px,b}$
$dim$	Kantenlänge in Pixeln
$G$	Geographischer Zielbereich der Analyse
$k_{t,b}$	Kachel des Zielbereichs $G$ von Spektralband $b \in B$ zum Aufnahmezeitpunkt $t \in T$
$K_b$	Menge aller Kacheln $k_{t,b} \in K_b$ mit identischem Spektralband $b$
$pts_{pos\_px,b}$	Pixelzeitreihe des Pixels an Position $pos\_px$ des Bandes $b$
$S$	Menge der Szenen $s_t \in S$ der Eingabedatenmenge
$T$	Menge der Aufnahmezeitpunkte $t \in T$
$tk_{t,pos\_tk,b}$	Teilkachel mit dem Aufnahmezeitpunkt $t$ , der Teilkachelposition $pos_{tk}$ und dem Band $b$
$TK_{pos\_tk,b}$	Menge aller Teilkachel mit der Teilkachelposition $pos_{tk}$ und dem Band $b$
$v_{t,pos\_px,b}$	Pixelwert für den Pixel an Position $pos\_px$ des Bandes $b$ zum Aufnahmezeitpunkt $t$

Tabelle 3.1: Beschreibung der für die Evaluation genutzten Testkonfigurationen



Dieser besteht aus den einzelnen nach  $t$  sortierten Szenen  $s_t$ , so wie in Abbildung 3.1 abgebildet.

Jede Kachel  $k_{t,b} \in K_b$  muss wie in Zeile 17 beschrieben in eine individuell festgelegte Anzahl von kleineren Kacheln, sogenannte Teilkacheln  $tk_{t,pos\_tk,b}$ , zerschnitten werden. Dabei werden zunächst aus Gründen der Vereinfachung nur quadratische Teilkacheln betrachtet. Dies impliziert, dass die Dimensionen der Kacheln  $k \in K$  so definiert werden müssen, dass eine restlose Aufteilung in Teilkacheln möglich ist. Theoretisch ist aber auch die Nutzung von allgemein rechteckigen Teilkacheln möglich. Nachdem alle Kacheln  $k_{t,b} \in K$  in Teilkacheln  $tk_{t,pos\_tk,b}$  geschnitten wurden, werden diese gemäß ihrer geographischen Position sowie ihres Spektralbands gruppiert, so dass die Gleichung 3.1 gilt.

$$tk_{t,pos\_tk,b} \in TK_{pos\_tk,b} \forall tk_{t,pos\_tk,b} \quad (3.1)$$

Die geographische Position ist in Abbildung 3.1 anhand der Tupel der Teilkacheln markiert, das als Wert für das Attribut  $pos\_tk$  fungiert. Durch die Gruppierung aller Teilkacheln in Zeile 18 entstehen  $n \times b$  verschiedene Gruppen von Teilkacheln für  $n = \{\text{Anzahl der Teilkacheln pro Kachel}\}$  und  $b = \{\text{Anzahl der betrachteten Spektralbänder}\}$ . Im in Abbildung 3.1 gezeigten Beispiel werden drei Kacheln eines Bandes  $b$  in jeweils 36 Teilkacheln unterteilt. Da  $b = 1$  ergeben sich gemäß der Formel 3.1 36 Gruppen von Teilkacheln. Dabei sind alle Teilkacheln einer Gruppe  $TK_{pos\_tk,b}$  durch das identische Positionstupel gekennzeichnet.

Jede dieser Gruppen wird dann in Zeile 22 nach dem jeweiligen Aufnahmedatum  $t$  der einzelnen Teilkacheln  $tk_{t,pos\_tk,b} \in TK_{pos\_tk,b}$  sortiert.

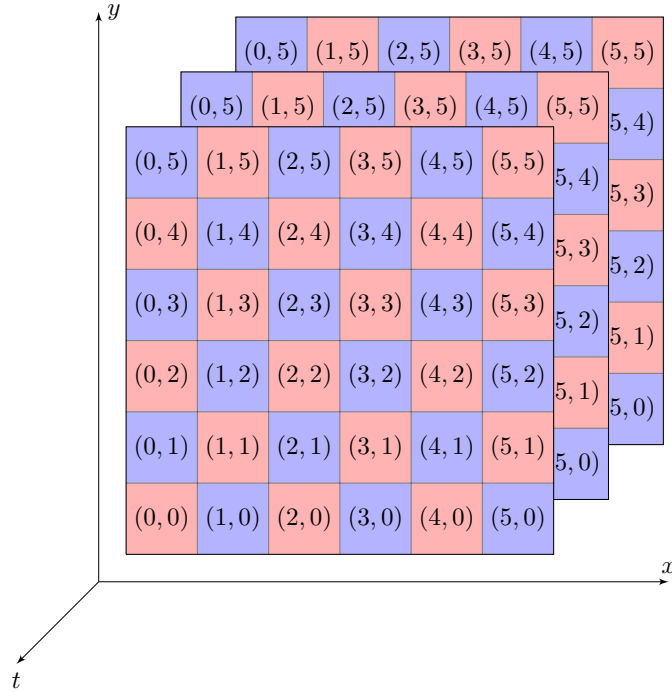


Abbildung 3.1: Beispielhafte Kacheln  $k_{t,b} \in K$  mit gleicher Kantenlänge  $\dim_k$ , ausgeschnitten aus Szenen  $s_t \in S$ . Die Tupel geben die Position  $pos\_tk$  der Teilkacheln  $tk_{t,pos\_tk,b}$  der Kachel an und werden als Gruppenschlüssel zum Gruppieren der Teilszenen verwendet.

$$pts_{pos\_px,b} = \langle px_{t_1,pos\_px,b}, \dots, px_{t_i,pos\_px,b}, \dots, px_{t_n,pos\_px,b} \rangle \mid t_1, \dots, t_i, \dots, t_n \in T, b \in B \quad (3.2)$$

Um die Veränderung des Vegetationsindex jedes betrachteten Pixels analysieren zu können, muss in Zeile 26 zunächst für jeden Pixel eine Pixelzeitreihe  $pts_{pos\_px,b}$  gebildet werden, die wie in Gleichung 3.2 definiert strukturiert sind. Da die Gruppen der Teilkacheln  $TK_{pos\_tk,b}$  nur die Pixelwerte eines Spektralbandes enthalten, gilt für alle Pixelzeitreihen implizit, dass sie nur die Pixelwerte eines Spektralbandes enthalten. Folglich muss bei der Konstruktion der Pixelzeitreihen keine weitere Unterscheidung nach diesem Kriterium erfolgen. In jeder Pixelzeitreihe  $pts_{pos\_px,b} \in TK_{pos\_tk,b}$  ist jedem bekannten Vegetationswert  $v_{t,pos\_px,b}$  genau ein fester Zeitpunkt  $t \in T$  zugeordnet, zu dem die Ausprägung des Werts durch die Auswertung des Pixelwerts des Satellitenbildes bekannt ist. Allerdings ist nicht jedem Zeitpunkt  $t$  für alle Positionen  $pos\_px$  eines Bandes  $b$  ein Wert  $v_{t,pos\_px,b}$  zugeordnet, da die Eingabedaten entweder keine Szene  $s_t$  enthalten oder der Pixel an der Stelle  $pos\_px$  für das betrachtete Spektralband  $b$  keinen validen Vegetationsindexwert enthält. Dies kann aufgrund externer Störfaktoren wie Wolken, Wolkenschatten oder Schnee zum Zeitpunkt der Aufnahme der Fall sein. Sollte der Wert für  $v_{t,pos\_px,b}$  nicht verfügbar sein, wird der Aufnahmezeitpunkt  $t$  aus der entsprechenden Pixelzeitreihe entfernt. Auf Basis der vorhandenen Werte wird dann in Zeile 31 mittels einer Regressionsanalyse ein Koeffizientenvektor  $cv_{pos\_px,b}$  für jede Pixelzeitreihe  $pts_{pos\_px,b}$  approximiert. Dieser wird durch die Werte  $v_{t,pos\_px,b} \forall t \in T$  der entsprechenden Pixelzeitreihe bestmöglich annähert. Für die Regressionsanalyse wird das Regressionsverfahren Support Vektor Regression eingesetzt, das nachfolgend erklärt wird.

**Algorithm 1** Analyse der Vegetationsveränderung einer geographischen Region

---

```

1: function APPROXIMATEVEGETATIONINDEX( $S, G, dim_{tk}$ )
2:   input: Landsat-Szenen  $S$ , Zielregion  $G$ , Kantenlänge der Teilkacheln  $dim_{tk}$ 
3:   output: Koeffizientenvektor  $cv_{pos\_px,b}$  für jeden Pixel  $px \in k \forall k \in K$ 
4:   ▷ Gruppieren alle Szenen gemäß dem Aufnahmezeitpunkt
5:   for  $s \in S$  do
6:      $S_t \leftarrow s_t$ 
7:   end for
8:   ▷ Schneide den Zielbereich aus den Szenen und gruppieren alle Kacheln gemäß ihrem Band
9:   for all  $S_t$  do
10:    for all  $s_t \in S_t$  do
11:       $k_{t,b} \leftarrow matching\_Region(s_t, G)$ 
12:       $K_b \leftarrow k_{t,b}$ 
13:    end for
14:  end for
15:  ▷ Schneide alle Kacheln in Teilkacheln und gruppieren alle Teilkacheln gemäß ihrer Position und ihrem Band
16:  for all  $k_{t,b} \in K_b$  do
17:    Teilkacheln  $tk_{t,pos\_tk,b} \leftarrow zerschneide\_Kachel(k_{t,b})$ 
18:     $TK_{pos\_tk,b} \leftarrow tk_{t,pos\_tk,b}$ 
19:  end for
20:  ▷ Sortieren die Teilkachelgruppen nach Aufnahmezeitpunkt
21:  for all  $TK_{pos\_tk,b}$  do
22:     $TK_{pos\_tk,b} \leftarrow sortiere \forall tk_{t,pos\_tk,b} \in TK_{pos\_tk,b} \text{ nach } t$ 
23:    ▷ Erstelle jeweils eine Pixelzeitreihe für jede Pixelposition in einer Teilkachelgruppe
24:    for all  $tk_{t,pos\_tk,b} \in TK_{pos\_tk,b}$  do
25:      for all  $px_{t,pos\_px,b} \in tk_{t,pos\_tk,b}$  do
26:        Pixelzeitreihe  $pts_{pos\_px,b} \leftarrow pts_{pos\_px,b}.append(v_{t,pos\_px,b})$ 
27:      end for
28:    end for
29:    ▷ Approximieren alle Pixelzeitreihen der Teilkachelgruppe
30:    for all  $pts_{pos\_px,b}$  do
31:      Koeffizientenvektor  $cv_{pos\_px,b} \leftarrow approximiere v_{t,pos\_px} \forall t \in pts_{pos\_px,b}$ 
32:    end for
33:  end for
34: end function

```

---

### 3.1.1 Support Vektor Regression

Die Support Vector Regression ist ein Regressionsverfahren, das auf *Support Vector Machines*, auch Stützvektormaschinen genannt, aufbaut [DBK<sup>+</sup>97]. Support Vector Machines bezeichnen ein mathematisches Verfahren zur Mustererkennung, das seine Ursprünge im maschinellen Lernen hat.

$$\{(x_1, y_1), \dots, (x_n, y_n)\} \subset \chi \times \mathbb{R} \quad (3.3)$$

Dabei wird für einen Trainingsdatensatz gemäß der Formel 3.3, wobei  $\chi$  die räumliche Dimension der Eingabedaten angibt, für jeden Beobachtungswert  $y_i$  ein Wert  $\hat{y}_i$  bestimmt. Die Distanz  $\epsilon$  zwischen  $y_i$  und  $\hat{y}_i$  wird durch  $\epsilon_i = |y_i - \hat{y}_i|$  berechnet. Um die Güte der Regression anzupassen, kann eine maximale Distanz  $\bar{\epsilon}$  definiert werden. Bei der Annäherung der Funktion  $f(x)$  anhand der Beobachtungswerte  $y$  werden nur jene Werte  $y_i$  mit  $\epsilon_i \leq \bar{\epsilon} \forall i$  berücksichtigt. Dadurch wird eine

übermäßige Verzerrung der Regressionsfunktion durch statistische Ausreißer verhindert.

$$\min_{\omega} |\omega| = \sum_{i=0}^n \epsilon_i \quad (3.4)$$

Um eine optimale Annäherung der Funktion  $f(x)$  und somit der approximierten Werte an die tatsächlichen Werte zu erhalten, muss  $f(x)$  so gewählt werden, dass die Summe der Abweichungen gemäß Formel 3.4 minimal ist.

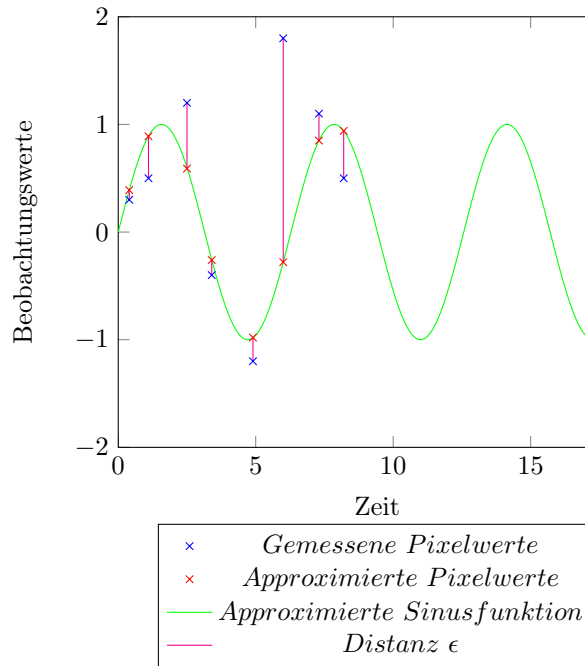


Abbildung 3.2: Graphische Darstellung der Berechnung der Distanz  $\epsilon$  zwischen gemessenen und approximierten Werten

Im Rahmen dieser Bachelorarbeit wird die SVR-Methode auf die einzelnen Pixelzeitreihen  $pts_{pos\_px,b}$  angewendet. Dadurch wird auf Basis aller  $P_{t,pos\_px,b} \in pts_{pos\_px,b}$ , für die ein Pixelvegetationsindexwert  $v_{t,pos\_px,b}$  vorliegt, ein Koeffizientenvektor berechnet, mit dessen Hilfe fehlende Werte der Reihe approximiert werden können. Außerdem werden alle Werte, die die maximale Distanz  $\bar{\epsilon}$  überschreiten, aus der Pixelzeitreihe gelöscht, da es sich höchstwahrscheinlich um nicht korrekte Werte handelt. Diese würden wenn sie weiterhin in der Pixelzeitreihe verbleiben würden, die Güte der Analyse verschlechtern, da diese alle Beobachtungswerte betrachtet und bei der Bildung der Approximierungsfunktion berücksichtigt.

Da die Güte der Approximierung durch bereits durchgeführte Experimente bereits bekannt und zufriedenstellend ist, kann auf einen Testdatensatz verzichtet werden.

### 3.1.2 Komplexitätsanalyse der Algorithmen

Da potenziell hunderte Satellitenbilder von möglicherweise großen Zielbereichen mithilfe des Algorithmus analysiert werden sollen, muss die Abhängigkeit der Algorithmen von der Eingabedatenmenge und den Eingabeparametern bestimmt werden. Um dies zu ermöglichen wird eine Komplexi-

tätsanalyse der Laufzeit des Schneide- sowie des Analysealgorithmus der Flink-Implementationen durchgeführt. Ebenso wird die Zeitkomplexität des Python-Programms bestimmt. Bei allen getätigten Komplexitätsanalysen soll der schlechtestmögliche Fall betrachtet werden, so dass eine obere Grenze für die Laufzeit des jeweiligen Algorithmus ermittelt wird, die mithilfe von Landau-Symbolen angegeben wird. Die Komplexität wird anhand der gegebenen Pseudocodes der Algorithmen veranschaulicht. Der Kommentar am Ende einer Zeile beschreibt dabei die jeweilige Zeitkomplexität der Zeile.

Da bereits vorhandene Schnittstellen verwendet wurden, um das Einlesen und Konstruieren der Kacheln der Daten zu implementieren werden nur die selbst umgesetzten Analysealgorithmen auf ihre Komplexität hin analysiert.

### Komplexitätsanalyse der Flink-Implementationen

Der in Algorithmus 2 angegebene Schneidealgorithmus wird von den beiden Flink-Varianten genutzt. Er zerteilt eine Kachel  $k_{t,b} \in K_b$  in  $n_{tk}$  Teilkacheln  $tk_{pos\_tk,b}$ . Da der Eingabeparameter des Programms nicht die Anzahl der Teilkacheln  $n_{tk}$  sondern die Kantenlänge  $dim_{tk}$  angibt, muss  $n_{tk}$  berechnet werden. Dabei wird in den Zeilen 5 und 6 jeweils die Anzahl der Teilkacheln pro Reihe beziehungsweise Spalte von Teilkacheln ermittelt, was in konstanter Zeit möglich ist. Diese beiden Werte erlauben es in den darauf folgenden Zeilen 8 und 9 über alle Teilkacheln  $tk_{t,pos\_tk,b} \in k_{t,b}$  zu iterieren, deren Objektattribute zu ermitteln und letztlich die entsprechen Teilkachelobjekte zu konstruieren. Aufgrund der Abhängigkeit der Schleifendurchläufe von der Anzahl der Teilkacheln pro Reihe und Spalte, wird eine Komplexität von  $\mathcal{O}(n_{tk})$  angenommen, da aufgrund der Annahme von quadratischen Kacheln und Teilkacheln  $nr_{tk} = ns_{tk}$  gilt. Der in Zeile 10 beginnende Block besteht aus Zuweisungen, die eine Komplexität von  $\mathcal{O}(1)$  besitzen. Lediglich die Schleife in Zeile 14 besitzt eine Zeitkomplexität von  $dim_{tk}$ . Da der innere Block der Schleife lediglich die Zuweisungen der Indexwerte enthält, die in konstanter Zeit ausführbar sind, gilt für die Schleife folglich eine Komplexität von  $\mathcal{O}(dim_{tk})$ . Daraus ergibt sich nach der Verkettung dieser Komplexität mit der Komplexität der Schleifen in den Zeilen 8 und 9 eine Gesamtkomplexität für den Schneidealgorithmus von

$$\mathcal{O}(nr_{tk} * ns_{tk} * dim_{tk}) = \mathcal{O}(n_{tk} * dim_{tk}) \quad (3.5)$$

Da die Kachel genau die zwei Dimensionen Breite und Höhe besitzt, gilt immer  $nr_{tk} * ns_{tk} = n_{tk}$ .

---

**Algorithm 2** Algorithmus zum Zerschneiden einer Kachel  $k_{t,b}$  in  $n_{TK}$  Teilkacheln  $tk_{t,pos\_tk,b}$ 


---

```

1: function CUTTILEINTOSUBTILES( $S, G$ )
2:   input: Kachel  $k_{t,b} \in K_b$ , Teilkachelkantenlänge  $dim_{tk}$ 
3:   output:  $n_{tk}$  Teilkacheln  $tk_{t,pos\_tk,b} \in TK_{pos\_tk,b}$ 
4:    $\triangleright$  Gruppieren alle Szenen gemäß dem Aufnahmezeitpunkt
5:   Berechne  $nr_{tk}$  als Anzahl der Teilkacheln pro Reihe  $r \in R$  von  $k_{t,b}$   $\triangleright 1$ 
6:   Berechne  $ns_{tk}$  als Anzahl der Teilkacheln pro Spalte  $s \in S$  von  $k_{t,b}$   $\triangleright 1$ 
7:    $\triangleright$  Iteriere über alle Teilkacheln der Kachel
8:   for all  $r_k \in R$  do  $\triangleright nr_{tk}$ 
9:     for all  $s_k \in S$  do  $\triangleright ns_{tk}$ 
10:       $pos_{tk} \leftarrow (r, s)$   $\triangleright 1$ 
11:       $tk = tk_{t,pos\_tk,b}$   $\triangleright 1$ 
12:       $Attr_{tk} \leftarrow Attr_k$   $\triangleright 1$ 
13:       $\triangleright$  Selektion der relevanten Vegetationsindexwerte
14:      for all  $r_{tk} < dim_{tk}$  do  $\triangleright dim_{tk}$ 
15:         $values_{tk} \leftarrow values_k$  des Zielbereichs  $\triangleright 1$ 
16:      end for
17:    end for
18:    Konstruktion des Teilkachelobjekts  $tk$  mit Attributen  $Attr_{tk}, pos_{tk}, values_{tk}$   $\triangleright 1$ 
19:  end for
20: end function

```

---

Nach dem Schneiden der  $n_s$  Kacheln in  $n_{tk}$  Teilkacheln pro Kachel, werden diese in Gruppen zusammengefasst und nach Aufnahmezeitpunkt sortiert. Das Gruppieren von  $n_s$  Teilkacheln  $tk_{t,pos\_tk,b}$  in  $n_{tk}$  Teilkachelgruppen  $TK_{pos\_tk,b}$  ist in  $\mathcal{O}(n_s)$  möglich. Für das Sortieren der Kacheln nutzt Flink eine Quicksort-Implementation. Obwohl die Worst-Case Komplexität von Quicksort  $\mathcal{O}(n^2)$  beträgt, kann stattdessen von  $\mathcal{O}(n * (\log(n)))$  ausgegangen werden, da Flink stattdessen einen Heapsort-Algorithmus verwendet, wenn die Nutzung des QuickSort-Algorithmus zu viele Rekursionen verursacht.

Wenn die Teilkacheln gruppiert und sortiert vorliegen, können sie mithilfe des in Algorithmus 3 angegebenen Algorithmus zu Pixelzeitreihen transformiert und dann analysiert werden. In der geschachtelten Schleife in Zeile 5 wird zunächst für jede Pixelzeitreihe  $pts_{pos\_px,b} \in TK_{pos\_tk,b}$  eine Hashmap erzeugt, die den Aufnahmezeitpunkt  $t$  als Schlüssel und den Vegetationsindexwert  $v_{t,pos\_px,b}$  als Wert nutzt. Darüber hinaus existiert eine Hashmap  $h$ , die für jede Pixelposition  $pos\_px$  die Hashmap der entsprechenden Pixelzeitreihe  $pts_{pos\_px,b}$  speichert. Es werden Hashmaps genutzt, da sowohl der Lese- als auch der Schreibzugriff auf ein Schlüssel-Wert-Paar, bei einer geeigneten Größe der Hashmap, eine Komplexität von  $\mathcal{O}(1)$  aufweist. Da zur Erzeugung aller Hashmaps über alle Pixelpositionen der Teilkacheln iteriert werden muss, ergibt sich die in Gleichung 3.6 angegebene Komplexität für die erste geschachtelte Schleife.

$$f_1 \in \mathcal{O}(dim_{tk} * dim_{tk}) = \mathcal{O}(dim_{tk}^2) \quad (3.6)$$

Nachdem alle benötigten Hashmaps erzeugt wurden, wird in Zeile 11 über alle Teilkacheln  $tk_{t,pos\_tk,b} \in TK_{pos\_tk,b}$  iteriert. Für jede dieser Teilkacheln muss nun jeder Pixelwert  $v_{t,pos\_px,b}$  der entsprechenden Pixelzeitreihe  $pts_{pos\_px,b}$  zugeordnet werden, wozu über alle Pixel iteriert werden muss. Dabei wird  $n_{v_{t,pos\_px,b}} \leq n_s$  angenommen, da nicht mehr Pixelwerte als Szenen vorhanden sein können. Da das Einfügen des Pixelwerts an der Position  $t$  in die Hashmap, die die Pixelzeitreihe  $pts_{pos\_px,b}$  speichert, in konstanter Zeit geschieht, besitzt der gesamte Codeblock von Zeile 11 bis 14 die in Gleichung 3.7 angegebene Komplexität.

$$f_2 \in \mathcal{O}(n_s * dim_{tk}^2) \quad (3.7)$$



Die in Zeile 19 beginnende Schleife durchläuft alle Pixelzeitreihen  $pts_{pos\_px,b} \in h$  der Teilkachelgruppe  $TK_{pos\_tk,b}$ . Um die SVR-Analyse auf die Pixelzeitreihe  $pts_{pos\_px,b}$  anwenden zu können, muss zunächst ein SVR-Problem  $svr_{pos\_px,b}$  definiert werden. Dazu werden in Zeile 20 zunächst für jeden Pixelwert  $v_{t,pos\_px,b} \in pts_{pos\_px,b}$  alle Werte  $t$  und  $v_{t,pos\_px,b}$  in die zwei Arrays *features* beziehungsweise *samples* gespeichert. Da hierfür jedes Schlüssel-Wert-Paar der Pixelzeitreihe ausgelesen werden muss, ist die Komplexität in  $\mathcal{O}(n_t k)$ . In der darauf folgenden Zeile ?? werden alle Werte des Arrays *samples* in den Wertebereich  $[-1, 1]$  projiziert. Da die Länge des Arrays  $n_T$  beträgt, besitzt auch diese Operation eine Komplexität von  $\mathcal{O}(n_t k)$ . Nachdem alle Daten für die SVR-Analyse aufbereitet wurden, kann nun in Zeile 22 das SVR-Problem konstruiert werden. Dazu müssen die Werte der Arrays *features* und *samples* in das SVR-Problem eingelesen werden. Demzufolge weist auch dieser Teil des Algorithmus eine Komplexität von  $\mathcal{O}(n_t k)$  auf.

Dass sowohl die Qualität als auch der Aufwand einer SVR maßgeblich von der Anzahl der Stützvektoren abhängen, schlägt sich in der Zeitkomplexität der SVR nieder, die  $\mathcal{O}(n_{features} * n_{samples}^3)$  beträgt. Da in unserem Fall  $n_{features} = n_{samples} = n_{tk}$ , gilt  $\mathcal{O}(n_{features} * n_{samples}^3) = \mathcal{O}(n_s^4)$ . Die theoretische Laufzeit der genutzten Analyse steigt also polynomiell mit der Anzahl der betrachteten Werte  $v_{t,pos\_px,b} \in pts_{pos\_px,b}$ . Da die polynomiale Komplexität die größte Komplexitätsklasse innerhalb des Schleifenkörpers ist, besitzt die Schleife die Komplexität  $f_3$ , die in Gleichung 3.8 beschriebene Komplexität.

$$f_3 \in \mathcal{O}(dim_s^2 * n_s^4) \quad (3.8)$$

Da  $f_1, f_2$  und  $f_3$  nacheinander ausgeführt werden, müssen die Komplexitäten durch die Gleichung 3.9 miteinander verkettet werden, um die Gesamtkomplexität des Algorithmus 3 zu bestimmen. Darüber hinaus wird dieser für jede Teilkachelgruppe durchgeführt, so dass  $\mathcal{O}(dim_{tk}^2 * n_s^4 * n_{tk})$  gilt.

$$f_1 = f_2 = f_3 = \mathcal{O}(dim_s^2 * n_s^4) \quad (3.9)$$

---

**Algorithm 3** Algorithmus zum Konstruieren und Analysieren von Pixelzeitreihen  $pts_{pos\_tk}$  einer Gruppe von Teilkacheln  $TK_{pos\_tk,b}$

---

```

1: function APPROXIMATEPIXELVALUES( $S, G$ )
2:   input: Nach  $t$  sortierte Teilkachelgruppe  $TK_{pos\_tk,b}$ , Teilkachelkantenlänge  $dim_{tk}$ 
3:   output: Koeffizientenvektoren  $cv_{pos\_px,b} \forall pts_{pos\_tk,b} \in TK_{pos\_tk,b}$ 
4:    $\triangleright$  Konstruiere eine Hashmap, die alle Pixelzeitreihen enthält
5:   for all  $x \in dim_{tk}$  do  $\triangleright dim_{tk}$ 
6:     for all  $y \in dim_{tk}$  do  $\triangleright dim_{tk}$ 
7:       Hashmap  $h \leftarrow h.put(pts_{pos\_px,b})$   $\triangleright 1$ 
8:     end for
9:   end for
10:   $\triangleright$  Iteriere über alle Teilkacheln der Teilkachelgruppe
11:  for all  $tk_{t,pos\_tk,b} \in TK_{pos\_tk,b}$  do  $\triangleright n_s$ 
12:    for all  $x \in dim_{tk}$  do  $\triangleright dim_{tk}$ 
13:      for all  $y \in dim_{tk}$  do  $\triangleright dim_{tk}$ 
14:         $pts_{pos\_px,b}.put(v_{t,pos\_px,b})$   $\triangleright 1$ 
15:      end for
16:    end for
17:  end for
18:   $\triangleright$  Approximiere alle Pixelzeitreihen  $pts_{pos\_px,b} \in h$ 
19:  for all  $pts_{pos\_px,b} \in h$  do  $\triangleright dim_{tk}^2$ 
20:    Definiere Variablen für das SVR-Problem  $svr_{pos\_px,b}$   $\triangleright n_s$ 
21:    Normalisiere Beobachtungswerte  $v_{t,pos\_px,b}$   $\triangleright n_s$ 
22:    Konstruiere SVR-Problem  $svr_{pos\_px,b}$   $\triangleright n_s$ 
23:    Führe die Regressionsanalyse für  $svr_{pos\_px,b}$  aus  $\triangleright n_s * n_s^3$ 
24:  end for
25: end function

```

---

Nachdem die einzelnen Komplexitäten für den in Flink implementierten Algorithmus ermittelt wurden, können diese zur Gesamtkomplexität des Algorithmus kombiniert werden. Diese muss die Komplexitäten für das Schneiden der Kacheln, für das Gruppieren und Sortieren der Teilkacheln sowie für die Analyse berücksichtigen. Wie in Formel 3.10 gezeigt, ist  $\mathcal{O}(dim_{tk}^2 * n_s^4)$  die Gesamtkomplexität der Analyse, die von den Flink-Implementationen durchgeführt wird. Da diese Analyse jedoch für jedes Spektralband  $b$  durchgeführt wird, ist die Komplexität des Programms  $\mathcal{O}(dim_{tk}^2 * n_s^4 * n_{TK} * n_b)$ . Da  $dim_{tk}^2 * n_{TK} = dim_K^2$  lässt sich die Komplexität auch als  $\mathcal{O}(dim_K^2 * n_s^4 * n_b)$  ausdrücken.

$$\mathcal{O}(n * (\log(n))) \in \mathcal{O}(n_s * dim_{tk}^2) \in \mathcal{O}(dim_{tk}^2 * n_s^4) \quad (3.10)$$

### Komplexitätsanalyse der Python-Implementation

Im Gegensatz zu den Flink-Varianten besteht das Python-Programm lediglich aus einem Algorithmus zum konstruieren und approximieren von Pixelzeitreihen der Pixel eines Zielbereichs aus  $n_s$  Szenen  $s_{t,b} \in S_b$ . Dieser ist in Algorithmus 4 angegeben. Der Zielbereich wird durch Kacheln  $k_{t,b} \in K_b$  mit der Kantenlänge  $dim_k$  beschrieben. Im Gegensatz zu den beiden anderen Varianten müssen die Kacheln nicht quadratisch sein, es wird aber aus Gründen der Vergleichbarkeit angenommen. Der Algorithmus ist dabei in drei Phasen  $p_1, p_2$  und  $p_3$  eingeteilt, die hintereinander ausgeführt werden. Die erste Phase dient lediglich dazu die Pixelzeitreihendatenstruktur in Form von verketteten Listen bzw. Sets zu konstruieren. Da für jeden Pixel der zu analysierenden  $px_{pos\_px,b} \in K_b$  eine Pixelzeitreihe  $pts_{pos\_px,b}$  in konstanter Zeit konstruiert wird, gilt

$$f(p_1) \in \mathcal{O}(n_b * dim_k^2). \quad (3.11)$$

In der zweiten Phase werden die Pixelzeitreihen mit den entsprechenden Vegetationsindexwerten  $v_{t,pos\_px,b}$  in konstanter Zeit befüllt. Dies geschieht für jeden Pixel einer jeden Kachel  $k_{t,b} \in K_b \forall b \text{ in } B$ . Somit beträgt die Komplexität für Phase 2

$$f(p_2) \in \mathcal{O}(n_b * n_s * \dim_k^2). \quad (3.12)$$

In der dritten Phase werden schließlich die konstruierten und befüllten Pixelzeitreihen mittels der Support Vektor Regression analysiert. Dazu wird zunächst mithilfe der drei Schleifen in den Zeilen 5 bis 7 über alle Pixelzeitreihen iteriert. In Zeile 10 werden alle nicht validen Werte aus der Pixelzeitreihe entfernt, da sie sonst die Analyse verfälschen könnten. Da das Entfernen der Werte eine Zeitkomplexität von  $\mathcal{O}(n)$  aufweist, wird stattdessen ein neues Array aus validen Werten konstruiert, da für die Anfügen-Operation  $\mathcal{O}(1)$  gilt. Somit gilt für diese Schleife eine Gesamtkomplexität von  $\mathcal{O}(n_s)$ , da die Anzahl der Werte in der Pixelzeitreihe maximal der Anzahl der Szenen entspricht. Im Gegensatz dazu verfügt die Regressionsfunktion in Zeile 13 über eine Komplexität von  $n_s * n_s^3$ . Demzufolge ergibt sich nach der Verkettung der Komplexitäten der Zeilen 5 bis 13 eine Komplexität für Phase 3 von

$$f(p_3) \in \mathcal{O}(n_b * \dim_k^2 * (n_s + n_s * n_s^3)) = \mathcal{O}(n_b * \dim_k^2 * n_s^4). \quad (3.13)$$

$$f(ph_1) \in o(f(ph_2)) \in o(f(ph_3)) \quad (3.14)$$

Da sich aus den Komplexitäten, die durch die Gleichungen 3.11, 3.12 und 3.13 angegeben sind, gemäß Gleichung 3.14 ergibt, dass  $f(p_3)$  die größte Komplexität besitzt, reicht diese aus, um die Komplexität des gesamten Python-Programms zu beschreiben.

Beim Vergleich der Komplexitäten der Flink- und der Python-Versionen des Programms, fällt auf, dass  $\mathcal{O}(n_b * \dim_k^2 * n_s^4) = \mathcal{O}(n_b * \dim_k^2 * n_s^4)$ . Die theoretische Laufzeit aller drei Implementationsvarianten ist also identisch.

---

**Algorithm 4** Python-Algorithmus zur Konstruktion und Approximierung von Pixelzeitreihen  $PTS_{pos,b}$

---

```

1: function CUTTILEINTOSUBTILES( $S, G$ )
2:   input:  $n_B$  Gruppen von Szenen  $s_{t,b} \in S_b | b \in B, \dim_K$ 
3:   output: Koeffizientenvektoren  $cv_{pos\_px,b} \forall pts_{pos\_tk,b} \in TK_{pos\_tk,b}$ 
4:   ▷ Iteriere über jede Pixelzeitreihe
5:   for all  $b \in B$  do                                     ▷  $n_B$ 
6:     for all  $x \in \dim_K$  do                               ▷  $\dim_K$ 
7:       for all  $y \in \dim_K$  do                             ▷  $\dim_K$ 
8:          $pos \leftarrow x, y$                                ▷ 1
9:         ▷ Konstruiere valide Arrays zur Berechnung der Analyse
10:        for all  $v_{pos\_px,t,b}$  in  $pts_{pos\_px,b}$  do           ▷  $n_S$ 
11:          Entferne ungültige Werte  $v_{pos\_px,t,b}$  aus  $pts_{pos\_px,b}$  ein ▷ 1
12:        end for
13:        Führe die Regressionsanalyse für  $pts_{pos\_px,b}$  aus   ▷  $n_s * n_s^3$ 
14:      end for
15:    end for
16:  end for
17:  Konstruktion des Teilkachelobjekts  $tk_{t,pos\_tk,b}$  mit Attributen  $Attr_{tk}, pos_{tk}, values_{tk}$  ▷ 1
18: end function

```

---

## 3.2 Umsetzung des Algorithmus mit Apache Flink

Der in Unterkapitel 3.1 beschriebene Algorithmus wird im Rahmen dieser Arbeit 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-Programmiererschnittstelle 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 Implementationen ist der vom Geographischen Institut der Humboldt-Universität zu Berlin umgesetzte Algorithmus, der in Python implementiert ist.

Wie bereits in Kapitel 2.2.3 beschrieben, sind Flink-Programme unabhängig von der verwendeten Programmierschnittstelle Datenströme, die aus Datenquellen, Datentransformationen und Datenausgaben bestehen. Der entsprechende Datenfluss-Graph, den das Flink-Programm beschreibt, ist in Abbildung 3.3 abgebildet und wird für beide Implementationsvarianten verwendet. Die im nachfolgenden Abschnitt referenzierten Arbeitsschritte beziehen sich auf die mit den entsprechenden Nummern gekennzeichneten Datentransformationen. 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. Diese werden mithilfe eines Parsers eingelesen, der die einzelnen Szenen in Datenobjekte umwandelt. Diese verfügen über alle Attribute, die zur Beschreibung der Szene notwendig sind, darunter, das Aufnahmedatum, die Eckkoordinaten sowie ein Array der Vegetationsindexwerte. Um die Szenen für die spätere Analyse aufzubereiten, werden die erzeugten Datenobjekte in Verarbeitungsschritt Nummer 2 gemäß ihrem jeweiligen Aufnahmedatum gruppiert. Diese Gruppierung erfolgt durch die Anwendung der *GroupBy*-Transformation, die alle Szenen als Eingabemenge hat und gemäß dem Schlüsselwert Aufnahmezeitpunkt  $t$  in Gruppen  $S_t | t \in T$  zusammengefasst ausgibt. Aus allen so gruppierten Szenen werden dann in Schritt 3 mithilfe einer *groupReduce*-Funktion Kacheln zurecht geschnitten, die nur noch die für den durch den Nutzer bestimmten geographischen Bereich relevanten Teilszenen beinhalten. Aktuell muss dieser Bereich eine kleinere geographische Region abbilden als die Szenen der Eingabedatenmenge. Diese werden dabei außerdem gemäß ihrem Spektralband gruppiert, so dass die Schleife in Zeile 9 des Algorithmus 1 komplett implementiert ist. Die Kachel-Datenobjekte sind dabei identisch zu den Szenen-Datenobjekten aufgebaut, da eine Kachel lediglich eine Szene mit anderen Abmessungen ist. Für diese Arbeitsschritte wird eine bereits vorhandene Implementation verwendet. Nachdem die Ausgangsszenen auf den relevanten geographischen Bereich reduziert wurden, werden alle Kacheln  $k_{t,b} \in K_b$  in kleinere Teilkacheln  $tk_{t,pos\_tk,b}$  zerschnitten. Dafür wird wie in Schritt 4 dargestellt der *Flatmap*-Operators auf jede Kachel angewendet. Da alle Kacheln unabhängig voneinander verarbeitet werden, können alle FlatMap-Operationen in diesem Schritt parallelisiert ausgeführt werden. Die Teilkacheln werden dabei als Teilkachel-Datenobjekte umgesetzt, die alle Eigenschaften der Kachel-Objekte implementieren und zusätzlich über ein Positionsattribute verfügen. Damit alle Pixel in genau einer Teilkachel enthalten sind, muss die Bedingung 3.15 gelten. Wäre dies nicht der Fall, würden bei der Anwendung der FlatMap-Operation Pixel verloren gehen. Dies bedeutet im Umkehrschluss, dass die Kantenlängen der ungeschnittenen Kacheln mindestens einen ganzzahligen Teiler besitzen müssen. Eine Alternative wäre das Zulassen von Rest-Teilkacheln, dann wäre die Bedingung 3.15 nicht mehr notwendig. Das Erzeugen der Teilkacheln entspricht der Schleife in Zeile 16 des Algorithmus 1.

$$\text{Kantenlänge}_{Kachel} \bmod \text{Kantenlänge}_{Teilkachel} = 0 \quad (3.15)$$

Nachdem die Teilkacheln  $tk_{t,pos\_tk,b}$  erzeugt wurden, werden sie wiederum gemäß ihrem Zeitstempel  $t$  in Teilkachelgruppen  $TK_{pos\_tk,b}$  gruppiert. Anschließend wird der *Sort*-Operator genutzt,

um alle Teilkacheln in den Teilkachelgruppen jeweils nach dem Aufnahmedatum  $t$  der enthaltenen Teilkacheln zu sortieren. Dies ist notwendig, um die korrekte zeitliche Abfolge der Beobachtungen sicherzustellen, die bei der Analyse der Daten vorausgesetzt wird. Die Gruppierung und Sortierung entsprechen den Arbeitsschritten 5 und 6. Nachdem die Teilkacheln auf diese Weise vorbereitet wurden, können die Pixelzeitreihen  $pts_{pos\_tk}$ , wie in den Zeilen 24 bis 28 beschrieben, konstruiert werden. Diese erfolgt zusammen mit der Analyse durch die Anwendung des GroupReduce-Operators auf alle nach  $t$  sortierten Gruppen von Teilkacheln  $TK_{pos\_tk,b}$  wie in Schritt 7 beschrieben. Nachdem innerhalb der GroupReduce-Funktion die Pixelzeitreihen für die verarbeitete Teilkachelgruppe konstruiert wurden, wird die Analyse auf diese angewendet. Dazu wird wie in Kapitel 3.1 beschrieben ein SVR-Modell konstruiert und gelöst, so dass die Vegetationsindexwerte  $V_{pos\_px,t}$  entsprechend approximiert werden.

Bei der Durchführung beider Analysen werden sowohl bei der Nutzung der Java- als auch bei der Nutzung der Python-Programmierschnittstelle von Flink teilweise externe Bibliotheken genutzt. Die Unterschiede bei der Nutzung der jeweiligen Bibliotheken werden in Sektion 3.2.1 aufgezeigt.

Nach der Durchführung der Analyseoperationen werden die Analyseergebnisse in Form eines 4-Tupels der Form  $(Pixelposition_x, Pixelposition_y, Band, Ergebnisvektor \alpha)$  gemäß Schritt 8 in einer Textdatei gespeichert und in einer vom Nutzer zu spezifizierenden Ausgabedatei abgelegt.

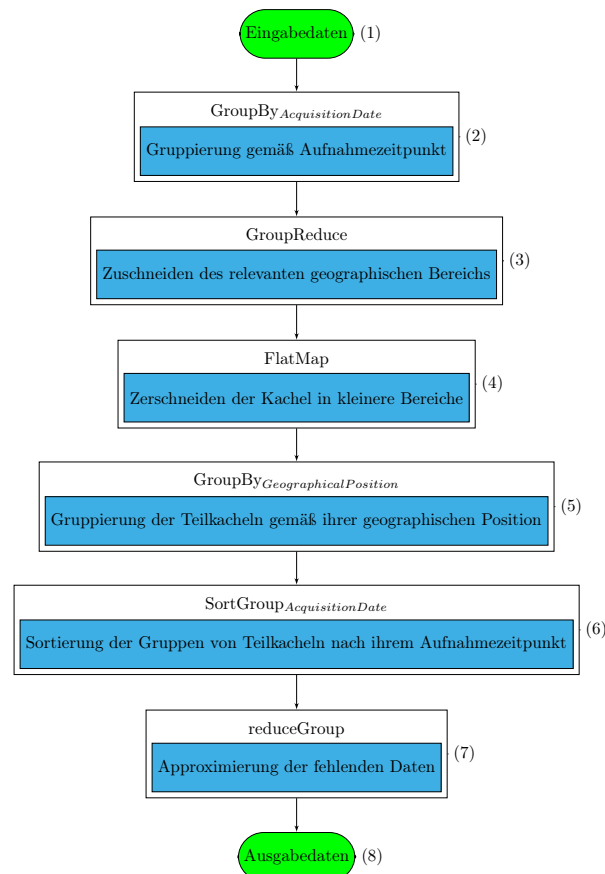


Abbildung 3.3: Visualisierung des Datenstroms des Analysealgorithmus in Flink

### 3.2.1 Unterschiede zwischen den drei Implementationsvarianten

Die Implementation des Algorithmus als Apache Flink-Programm erfolgt über die Nutzung der jeweiligen Programmierschnittstellen, die für Java und Python zur Verfügung stehen. In beiden Varianten wird dabei der Datenstrom aus Abbildung 3.3 umgesetzt. Aufgrund der Verschiedenheit der Sprachen Java und Python sowie der zugehörigen Schnittstellen sind die Implementationen nicht identisch, auch wenn sie prinzipiell denselben Algorithmus implementieren. Zusätzlich unterscheidet sich auch der Funktionsumfang der beiden Schnittstellen, was teilweise durch die unterschiedlichen Spracheigenschaften bedingt ist. Diese Unterschiede müssen bei der Implementation berücksichtigt werden.

Ein bedeutender Unterschied zwischen der Java- und der Python-Programmiererschnittstelle ist die unterschiedliche Art des effizienten Speicherns von Objekten. Flink Java-Objekte werden mithilfe eingebauter Serialisierer beziehungsweise durch die Nutzung von Kryo in Byte Arrays umgewandelt, um die angelegten Objekte effizient im Speicher halten zu können. Für Python-Objekte müssen hingegen für jedes Objekt eigene Serialisierer implementiert werden, die die Python-Objekte in Java-Objekte umwandeln. Diese werden von Flink dann entsprechend serialisiert. Entgegen der Spracheigenschaft von Python, die eine dynamische Typisierung von Variablen vorsieht, müssen bei der Serialisierung alle Attribute der Objekte statisch typisiert sein. Auf diese Weise wird den Attributen eine Typinformation beigelegt, die es Flink erlaubt den Datentyp des Attributes zu erkennen. Basierend auf dieser Typinformation weist Flink bei der Speicherung der Daten jedem Attribut Speicherplatz in der dem Datentyp entsprechenden Größe zu. Da von der Python-Schnittstelle zur Zeit lediglich die Datentypen Integer, Float, Boolean, String sowie Byte Arrays unterstützt werden, müssen andere Datentypen vor der Serialisierung in akzeptierte Typen umgewandelt werden. Des Weiteren muss für jeden Serialisierer auch ein Deserialisierer vorhanden sein, der die Java-Objekte wieder in Python-Objekte transformiert.

Wie bereits in Sektion 2.2.4 erwähnt, verfügt Python über die Möglichkeit Programmteile direkt in C auszuführen, was die Verarbeitungsgeschwindigkeit der betreffenden Abschnitte des Algorithmus steigern kann. Dementsprechend wird unter anderem für das Einlesen der Daten und die Regression die Python-Bibliotheken NumPy, GDAL und scikit-learn genutzt, die teilweise in C geschrieben und lediglich in Python eingebettet ist. Es ist jedoch nicht sicher, ob dies ausreicht, um den Algorithmus schneller auszuführen als die entsprechenden Java-Implementation, da diese die Just-in-time- Kompilierung unterstützt, die während der Ausführung des Java-Programms Optimierungspotenzial durch Caching und ähnliche Techniken erkennt und entsprechend nutzt. Es soll Ziel der Untersuchungen dieser Bachelorarbeit sein, welche Variante bei der Anwendung des in Kapitel 3.1 beschriebenen Algorithmus schneller ausgeführt wird.

Ein weiterer Unterschied besteht zwischen der Ausgabe der Java-Flink- und der PyFlink-Variante. Während Flink eine Java-Programmiererschnittstelle zur Verfügung stellt, die es erlaubt die Ergebnisse der parallelisiert ausgeführten Transformationen in eine Datei zu schreiben, ist dies für die Python-Schnittstelle nicht möglich. Dementsprechend werden die Ergebnisse der PyFlink-Variante in  $n$  Textdateien geschrieben, wobei  $n$  dem Grad der Parallelisierung des Programms entspricht.

Zusätzlich zu den Implementationen des Algorithmus in Apache Flink mittels der Java- bzw. der Python-Programmiererschnittstelle wird eine weitere Implementierung in reinem Python betrachtet. Ähnlich wie die PyFlink-Variante besitzt diese die in Sektion 2.2.4 beschriebenen Vor- und Nachteile eines Python-Programms. Sie nutzt außerdem dieselbe Bibliothek für die Durchführung der Regression. Aufgrund der mangelnden Parallelisierbarkeit dieser Implementationsvariante nutzt sie auch eine veränderte Version des Algorithmus. Aus den eingelesenen Szenen wird für jedes Spektralband  $b_i \in B$  lediglich auf den durch die Kacheln  $k_{t,b} \in K_b$  definierten Zielbereich  $G$  zugegriffen. Eine weitere Unterteilung in Teilkacheln erfolgt nicht, da diese bei einer sequentiellen Verarbeitung des Programms keinen Nutzen hätte. Zusätzlich würde die Schneidefunktion die Verarbeitungs-

zeit des Algorithmus erhöhen, so dass sie sich letztendlich nachteilig auswirken würde. Da also der gesamte Zielbereich aus  $n_B$  Kachelgruppen  $K_b$  besteht, wird der Analysealgorithmus auf diese angewendet.

# 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 diese 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 wird.

### 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 außen vorgelassen.

#### Skalierbarkeit

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 660 Megabyte ansteigt (bei einer Nutzung von Bildmaterial, das von Landsat 4 bzw. 5 aufgenommen wurde). Beim neueren Landsat 8 Satelliten kann eine Szene aufgrund von höherer Aufnahmequalität sogar eine Größe von bis zu 1,5GB besitzen. Da bei einer durchschnittlichen Überflugfrequenz einer geographischen Region von 16 Tagen durch Landsat 5 pro Jahr circa 22 Szenen pro Satellit von jeder dieser Regionen angefertigt werden, steigt die Datenmenge für jedes Jahr, das in die Analyse mit einbezogen werden soll, um circa 14,5 Gigabyte. Um eine Analyse des Vegetationsindex einer Region tätigen zu können, werden die Daten mehrerer Jahre verwendet, so dass die Eingabedatenmenge für den Algorithmus ein Vielfaches der 14,5 Gigabyte für jede Region betragen kann. Soll die Analyse auch ältere Daten des seit 1972 aktiven Landsat-Systems nutzen, erhöht sich die Eingabedatenmenge zusätzlich. Des weiteren ist es auch möglich, Aufnahmen mehrerer benachbarter Zielregionen zu aggregieren und so Szenen zu generieren, die das Vielfache der ursprünglichen 660 MB pro Szene aufweisen. Dies wäre notwendig, um größere



geographische Regionen zu untersuchen. Bei den Untersuchungen der Skalierbarkeit werden sowohl verschiedene Datenmengen als auch verschiedene Grade der Parallelisierung (für die betreffenden Varianten) betrachtet, um eine Abschätzung der Skalierbarkeit der drei Implementationsvarianten vornehmen zu können.

### Ausführungsgeschwindigkeit

Die Ausführungsgeschwindigkeit ist, wie in Sektion 2.2.5 beschrieben, als das Verhältnis von geleisteter Berechnungsarbeit pro Zeiteinheit definiert. Da diese bei identischen Parametern, zum Beispiel identischer genutzter Hardware, unverändert bleiben sollte, folgt daraus, dass sich die Ausführungszeit des Algorithmus verlängert, wenn die Datenmenge ansteigt. Zu lange Laufzeiten des Algorithmus sind jedoch bei der Nutzung zur Analyse von vielen geographischen Regionen von Nachteil. Deshalb sollen im Rahmen der nachfolgend beschriebenen Untersuchungen ermittelt werden, wie sich die Laufzeit verringern lässt, indem die Dimension der Kacheln sowie die Anzahl der Teilkacheln variiert wird.

## 4.2 Versuchsbeschreibung und -erwartungen

Um die drei Implementationsvarianten des Algorithmus anhand der in Sektion 4.1 erläuterten Evaluationskriterien beurteilen zu können, müssen alle Varianten auf dieselbe Testdatenmenge angewendet werden. Um zu verhindern, dass ein Ergebnis nicht repräsentativ ist, weil ein Versuch aufgrund von externen Einflüssen eine veränderte Laufzeit hat, werden von jeder Versuchskonfiguration drei Durchgänge ausgeführt und ein Durchschnittswert für alle erhobenen Messwerte ermittelt. Nach jedem Durchlauf wird das Cluster neu gestartet, um eventuelle Laufzeitbeschleunigungen durch Caching oder eine größere Menge temporärer Dateien voriger Tests zu verhindern und vergleichbare Rahmenbedingungen zu schaffen. Die Logdateien aller Versuchsdurchgänge sowie eine Datei mit den gesammelten Ergebnissen liegen der Arbeit auf CD bei.

Systemeigenschaft	Einzelmaschine des Clusters	Einzelmaschine Python-Evaluation
Anzahl CPUs	24	
Anzahl Kerne pro CPU	6	
Taktfrequenz Prozessorkern	2100 Mhz	
Arbeitsspeicher	128 GB	256 GB
Anzahl Threads	12	1
Betriebssystem	Ubuntu 14.04	

Tabelle 4.1: Eigenschaften der für die Evaluationstests genutzten Hardware.

Die technischen Eigenschaften der Maschinen, auf denen die verschiedenen Evaluierungen ausgeführt werden, sind in Tabelle 4.1 aufgeführt. Das Cluster besteht dabei aus 12 Einzelmaschinen des Clusters, so dass insgesamt 144 Threads parallel ausführbar sind. Das Cluster nutzt sowohl für den Zugriff auf die Eingabedaten als auch für die Speicherung der Ausgabedaten das verteilte Dateisystem HDFS. Im Gegensatz dazu liest die Einzelmaschine zur Python-Evaluation alle Daten von der lokalen Festplatte der Maschine ein. Als Betriebssystem aller Maschinen wird Ubuntu 14.04 genutzt. Darüber hinaus werden Apache Flink 0.9, Hadoop 2.6.0, Python 2.7 und Java 7 genutzt, um die Programme auszuführen.

Der komplette Datensatz, auf den die Implementationsvarianten angewendet werden, ist circa 242 Gigabyte groß und beinhaltet 373 Szenen. Dieser besteht aus Satellitenbildern, die von Landsat-Satelliten aufgenommen wurden und eine geographische Region in Brasilien abbilden. Die Nord-Süd-Ausdehnung der Region beträgt circa 1 Längengrad, also in etwa 211 Kilometer, die Ost-West-Ausdehnung circa 2 Breitengrade beziehungsweise circa 235 Kilometer. Insgesamt beinhaltet eine Szene aus dem Testdatensatz also eine Fläche von annähernd 50000 Quadratkilometern. Diese werden durch circa 8000 Pixel in der Breite und circa 7200 Pixel in der Länge dargestellt, wobei ein Pixel eine quadratische Fläche von 30 Metern Kantenlänge abbildet. Die Zeitstempel der Szenen reichen von Juni 1984 bis Dezember 2013. Alle Szenen enthalten jeweils die Daten der Spektralbänder 1, 2, 3, 4, 5 und 7.

Nachfolgend wird die Evaluierung der einzelnen Eingabeparameter Kachelkantenlänge  $dim_k$ , Teilkachelkantenlänge  $dim_{tk}$ , Anzahl der Szenen  $n_s$  sowie Grad der Parallelisierung  $dop$  beschrieben.

#### Auswirkung der Kachelkantenlänge $dim_k$ auf die Verarbeitungsgeschwindigkeit

Der erste Parameter, dessen Auswirkung auf die Verarbeitungsgeschwindigkeit bzw. die Laufzeit des Algorithmus ermittelt werden soll, ist die Kachelkantenlänge  $dim_k$ . Diese gibt die Größe der Kacheln in Pixeln an, die den Zielbereich der Analyse definieren. Aufgrund der Ergebnisse der Komplexitätsanalyse in Unterkapitel 3.1.2 wird erwartet, dass eine lineare Vergrößerung von  $dim_k$  die Laufzeit quadratisch ansteigen lässt. Bei der Python-Implementation besteht die Möglichkeit, dass sie aufgrund des limitierten Arbeitsspeichers eine maximale Kachelkantenlänge besitzt, da nicht mehr Pixelzeitreihen im RAM gespeichert werden können. Demzufolge wird für diese Variante keine gute Skalierbarkeit erwartet, während die parallelisierten Varianten vermutlich lediglich eine längere Ausführungszeit aufweisen werden, wenn  $dim_k$  vergrößert wird. Um die Erwartungen zu überprüfen wird eine Testreihe für alle drei Implementationsvarianten durchgeführt, die die anderen Parameter konstant hält, während verschiedene Werte für  $dim_k$  getestet werden.


#### Auswirkung der Teilkachelkantenlänge $dim_{tk}$ auf die Verarbeitungsgeschwindigkeit

Obwohl die Kantenlänge der Teilkacheln gemäß der Komplexitätsanalyse keine Auswirkung auf die Laufzeit des Algorithmus haben sollte, wird erwartet, dass sie die Verarbeitungsgeschwindigkeit der parallelisierten Varianten stark beeinflusst. Die Verarbeitungsgeschwindigkeit der Flink-Varianten wird vermutlich maßgeblich über die Anzahl der parallelisiert ausgeführten Threads des Programms bestimmt. Da die Kantenlänge der Teilkacheln die Anzahl der Teilkacheln pro Kachel und Band in Abhängigkeit von der Kachelkantenlänge gemäß Gleichung 4.1 definiert, können die Rechenkapazitäten bei  $n_{tk} < dop$  möglicherweise nicht vollständig genutzt werden. Dies führt möglicherweise zu einer geringeren Verarbeitungsgeschwindigkeit und in der Folge zu einer längeren Laufzeit. Aufgrund des in Abschnitt 3.2.1 beschriebenen Fehlens des Zerschneidens der Kacheln in Teilkacheln bei der Python-Implementation wird dieser Parameter nur für die Flink-Varianten evaluiert. Die Testreihe zur Überprüfung der Auswirkung der Teilkachelkantenlänge auf die Verarbeitungsgeschwindigkeit des Algorithmus wird mit konstanten weiteren Parametern durchgeführt.

$$n_{tk} = dim_k^2 / dim_{tk}^2 \quad (4.1)$$

#### Auswirkung der Anzahl der Szenen auf die Verarbeitungsgeschwindigkeit

Wie die Kachelkantenlänge  $dim_k$  ist auch die Anzahl der Szenen  $n_s$  ein Eingabeparameter, der entsprechend der Komplexitätsanalyse zumindest theoretisch die Laufzeit des Algorithmus beeinflusst. Auch bei der tatsächlichen Implementation wird erwartet, dass ein linearer Anstieg von  $n_s$  zu einer **sub-exponentiellen** Erhöhung der Laufzeit führt. Denn durch die Anzahl der Szenen wird

die Maximalanzahl der Elemente in den einzelnen Pixelzeitreihen definiert, die wiederum  Umfang der zugehörigen SVR-Analysen bestimmen. Deren Laufzeit steigt ~~sub-exponentiell~~ mit dem Faktor  $n_s^4$  bei linearem Anstieg der Anzahl der Szenen mit validen Werten, so dass eine Erhöhung von  $n_s$  eine starke Erhöhung der Laufzeit zur Folge haben sollte. Voraussetzung dafür ist, dass die Szenen genügend valide Pixelwerte enthalten, da sie sonst nicht Teil der Analyse sind und folglich auch nicht deren Komplexität beeinflussen. Bei der Python-Variante wird ebenso wie bei einer Erhöhung der Kachelkantenlänge die Existenz eines maximalen Werts für  $n_s$  angenommen, für den der verfügbare Arbeitsspeicher zur Verarbeitung aller Szenen ausreicht.

### Auswirkung des Grades der Parallelisierung auf die Verarbeitungsgeschwindigkeit

Ein Faktor, der bei der Komplexitätsanalyse komplett unberücksichtigt blieb, da er kein inhaltlicher Eingabeparameter des Algorithmus ist, ist der Grad der Parallelisierung *dop*. Wie bereits bei der Beschreibung der Evaluierung der Teilkachelkantenlänge erwähnt, wird erwartet, dass der Grad der Parallelität des Algorithmus die Laufzeit entscheidend beeinflussen kann. Ein zu geringer *dop* sorgt dafür, dass einige TaskSlots ungenutzt bleiben, während die genutzten TaskSlots größere Teilprobleme verarbeiten müssen. Andererseits kann ein zu hoher *dop* auch zu einem Mehraufwand führen, da mit steigendem *dop* auch der Kommunikationsaufwand für die Koordination der parallelisierten Instanzen steigt. Übersteigt dieser Aufwand die Einsparungen bei der Berechnung der Teilprobleme, ist eine obere Grenze erreicht, die den größten sinnvollen *dop* angibt. Dieses Kriterium wird nur für die Flink-Varianten evaluiert, da die Python-Implementation nicht parallelisiert wird. Um die Auswirkungen der Veränderung des Grades der Parallelisierung zu testen, werden alle anderen Parameter konstant gehalten, während für den *dop* verschiedene Werte im Bereich  $1 \leq \textit{dop} \leq \textit{Anzahl TaskSlots}$  angenommen werden.

Aufgrund des unterschiedlichen Funktionsumfangs und der damit einhergehenden ebenfalls unterschiedlichen Eingabeparameter sind streng genommen nur die PyFlink- und die Java-Flink-Varianten vergleichbar. Um einen Vergleich der Evaluationen der parallelisierten Implementationen und der Python-Variante zu ermöglichen, sind lediglich die Betrachtung der Evaluationen der Kachelkantenlänge  $\textit{dim}_k$  und der Anzahl der Szenen  $n_s$  möglich. Die anderen beiden Parameter  $\textit{dim}_{tk}$  und *dop* wurden jeweils so gewählt, dass sie eine schnelle Verarbeitung ermöglichen.


Es ist Ziel der Evaluierung diese Annahmen auf ihre Korrektheit zu überprüfen.

## 4.3 Auswertung

Im folgenden Abschnitt werden die Versuche für die verschiedenen Implementationsvarianten ausgewertet und miteinander verglichen. Wie beschrieben wurde dabei für alle drei Varianten der Einfluss der Variablen Kachelkantenlänge  $\textit{dim}_k$  und Anzahl der Szenen  $n_s$  auf die jeweilige Laufzeit und die Skalierbarkeit der einzelnen Varianten überprüft. Zusätzlich wurden für die parallelisierten Varianten der Einfluss der Teilkachelkantenlänge  $\textit{dim}_{tk}$  und des Grads der Parallelisierung *dop* überprüft. Diese Tests wurden auf den im vorigen Abschnitt beschriebenen Testsystemen durchgeführt.

### 4.3.1 Auswertung der Evaluierung der parallelisierten Varianten

#### Auswirkung der Teilkachelkantenlänge $\textit{dim}_{tk}$ auf die Verarbeitungsgeschwindigkeit

Die Ergebnisse für die Flink-Varianten zeigen, dass sowohl die Kantenlänge der Kacheln  $\textit{dim}_k$  als auch die Kantenlänge  $\textit{dim}_{tk}$  einen großen Einfluss auf die Ausführungsgeschwindigkeit haben. Wie in  Abbildung 4.1a zu sehen, bewegt sich die Laufzeit der verschiedenen Konfigurationen mit Kachelkantenlänge  $\textit{dim}_k = 500$ , identischer Anzahl der Szenen  $n_s = 373$  und einem Grad der

Parallelisierung  $dop = 144$  für variable Teilkachelkantenlängen  $dim_{tk}$  zwischen 18:52 Sekunden für die PyFlink-Variante mit  $dim_{tk} = 10$  und 6:00-00, muss noch bestätigt werden für die Java-Flink-Implementation mit  $dim_{tk} = 500$ . Dabei wirkt sich sowohl eine zu geringe als auch eine zu große Kantenlänge  $dim_{tk}$  negativ auf die Laufzeit der Implementationen aus, wobei die Java-Variante davon stärker betroffen ist. Am schnellsten ist der Algorithmus, wenn  $dim_k$  und  $dim_{tk}$  so gewählt werden, dass die Anzahl der Teilkachelgruppen  $n_{TK}$  in etwa dem angenommenen Grad der Parallelisierung entspricht. Die entsprechende Gleichung 4.1 zur Berechnung von  $n_{tk}$  ist in Formel 4.2 exemplarisch für die beiden Werte  $dim_{tk} = 20$  und  $dim_{tk} = 50$  gezeigt. Da für den angenommenen Wert  $dop = 144$   $100 < dop < 625$  gilt, erhöht sich die Laufzeit für Werte  $dim_{tk} < 20$  und  $dim_{tk} > 50$ . Basierend auf den Ergebnissen der Evaluierung scheint es optimal zu sein  $dim_{tk}$  so zu wählen, dass die Gleichung 4.3 gilt. Dies entspricht den Erwartungen bezüglich der optimalen Parallelisierung durch eine sinnvolle Wahl des Parameters  $dim_{tk}$  zum Wert  $dim_k$ .

$$\begin{aligned} n_{tk} &= 500^2 / 20^2 = 625 \\ n_{tk} &= 500^2 / 50^2 = 100 \end{aligned} \quad (4.2)$$

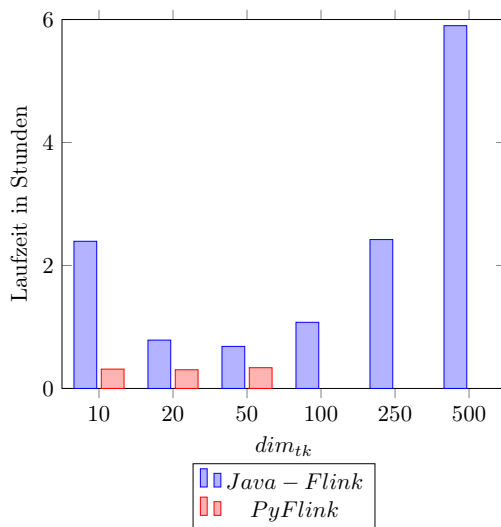
$$\min_{|n_{tk} - dop|} \quad (4.3)$$

Erklären lässt sich die längere Laufzeit für sehr kleine  $dim_{tk}$  mit dem erhöhten Aufwand für die Schneidefunktion sowie den größeren Aufwand für die Parallelisierung, weil mehr parallelisierbare Instanzen der Analysefunktion erzeugt werden müssen. Ist  $dim_{tk}$  dagegen zu groß gewählt, werden weniger Analysen als möglich parallel durchgeführt. Da die zu analysierende Datenmenge pro Instanz der Analysefunktion mit sinkender Anzahl der Teilkachelgruppen  $n_{TK}$  steigt, steigt die Laufzeit pro Instanz und demzufolge auch die Gesamtlaufzeit des Programms.

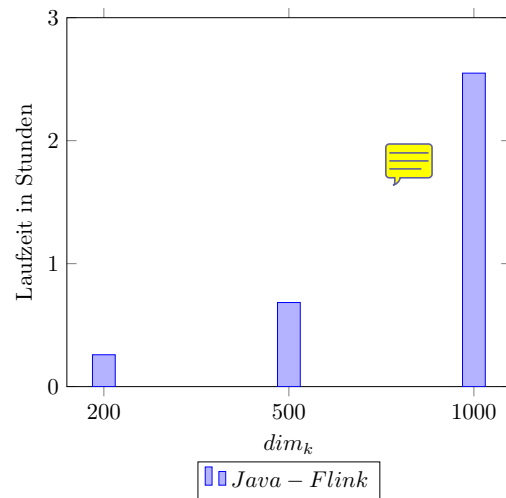
Es fällt auf, dass die PyFlink-Implementation für jede Testkonfiguration bedeutend schneller war als die äquivalente Java-Implementation. Vermutlich sind die Python-Funktionen zum Schneiden und Analysieren der Daten effizienter implementiert als in Java. Dies entspricht nicht der allgemeinen Erwartung, da sich PyFlink momentan noch in einem Beta-Status befindet. Dies macht sich insbesondere bei der Stabilität von PyFlink bemerkbar. So war es nicht möglich, Ergebnisse für  $dim_{tk} > 100$  zu ermitteln, da PyFlink bei dieser Laufzeit immer aufgrund von internen Kommunikationsfehlern abstürzte. Es wird jedoch auf Basis der ermittelten Ergebnisse für  $dim_{tk} \leq 100$  angenommen, dass die PyFlink-Implementation auch für größere Teilkachelkantenlängen eine geringere Laufzeit besitzt als die Java-Flink-Variante, sobald sie ähnlich stabil funktioniert.

### Auswirkung der Kachelkantenlänge $dim_k$ auf die Verarbeitungsgeschwindigkeit

Die Kachelkantenlänge  $dim_k$ , deren Auswirkung auf die Laufzeit der Java-Flink-Variante in Abbildung 4.1a dargestellt ist, verlängert die Laufzeit des Algorithmus für  $dim_k < 500$  zwar superlinear, jedoch nicht quadratisch. Für die Verdopplung von  $dim_k$  gilt dann jedoch quadratisches Wachstum, so wie es erwartet wurde. Das unerwartet geringe Wachstum der Laufzeit bei  $dim_k < 500$  ist nur auf den ersten Blick verwunderlich. Verantwortlich dafür ist die Struktur der genutzten Szenen. Diese sind als Rechteck in einem zweidimensionalen Array abgebildet, so dass die durch den Reihen- und Spaltenwert angegebene Position einer Zelle die Pixelposition innerhalb der Szene beschreibt. Der Inhalt der Zelle ist dann der Pixelwert. Im Original liegen die Szenen jedoch als gekippte Rechtecke vor, so dass ein in der Datenstruktur abbildbares Rechteck durch das Hinzufügen von Platzhaltern an den fehlenden Positionen erzeugt werden muss. Diese Platzhalter häufen sich in der linken oberen Ecke des Bildes, da die Originalszene meist im Uhrzeigersinn gekippt ist. Dadurch werden bei Programmausführungen mit kleiner Kachelkantenlänge größtenteils Pixelzeilen bestehend aus Platzhalterwerten erzeugt. Da diese Werte jedoch von der Analyse ausgeschlossen werden, um die Güte der Regression nicht negativ zu beeinflussen, werden dann zumeist gar keine Analysen



(a) Durchschnittliche Laufzeit der PyFlink und Java-Flink-Implementationen für  $dim_k = 500$  in Abhängigkeit von  $dim_{tk}$ .



(b) Durchschnittliche Laufzeit der Java-Flink-Implementierung für  $dim_{tk} = 50$  in Abhängigkeit von  $dim_k$ .

Abbildung 4.1: Laufzeit der Flink-Implementationen für variierende Teilkachelkantenlängen  $dim_{tk}$  und Kachelkantenlängen  $dim_k$  mit  $dop = 144$  und  $n_s = 373$ .

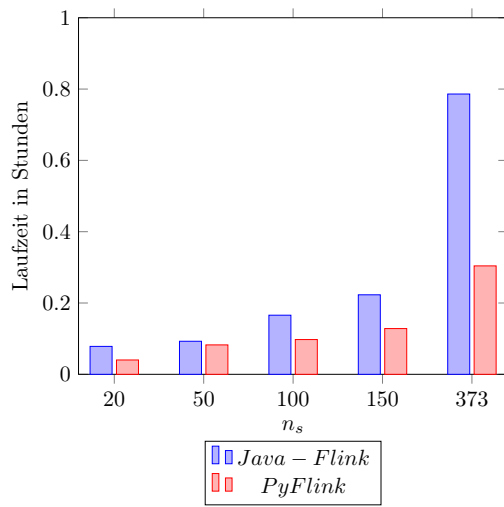
für die Pixelzeitreihen ausgeführt, so dass der Faktor  $dim_k$  den Anschein erweckt, die Laufzeit nicht entsprechend der Komplexitätsanalyse zu erhöhen. Bei der Betrachtung von größeren  $dim_k$  wird die quadratische Abhängigkeit der Laufzeit von der Kachelkantenlänge dann deutlicher, da sich der Anteil der Pixelzeitreihen, die valide Werte enthalten, sukzessive erhöht. Bei einer testweise Durchführung der Analyse einer Kachel in der Bildmitte wurde auch bei kleineren  $dim_k$  die erwartete Erhöhung der Laufzeit beobachtet.

Die quadratische Erhöhung der Laufzeit durch die Vergrößerung der Kachelkantenlänge  $dim_k$  wirkt sich negativ auf die Skalierbarkeit aus. Die maximale betrachtete Kachelgröße  $dim_k = 1000$  entspricht in der Realität gemäß der Abmessungen der Szenen des Testdatensatzes, der in Unterkapitel 4.2 beschrieben wurde, lediglich einer geographischen Region von circa 25 km Höhe und circa 30 km Breite. Sollen nun größere Zielregionen untersucht werden, würde die steigende Laufzeit die Analyse sehr langwierig und damit weniger praktikabel gestalten.

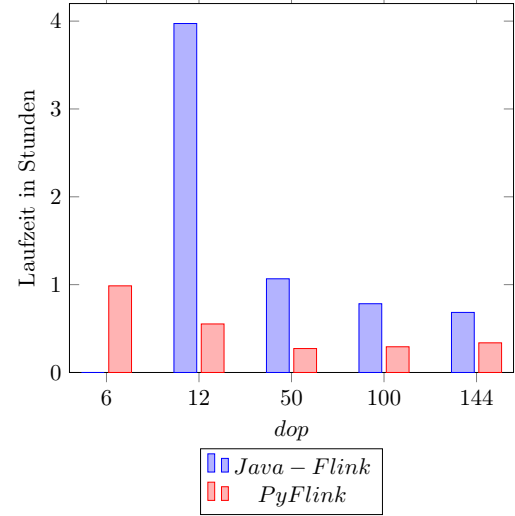
Für PyFlink konnte aufgrund der mangelhaften Stabilität des Systems keine Ergebnisse für die entsprechenden Werte für  $dim_k$  ermittelt werden.

### Auswirkung der Anzahl der Szenen $n_s$ auf die Verarbeitungsgeschwindigkeit

Die aufgrund der Komplexitätsanalyse erwartete **sub-exponentielle** Abhängigkeit von  $n_s$  hat sich bei den getätigten Messungen nicht bestätigt. Wie in Abbildung 4.2a zu sehen, ist der Anstieg der Laufzeit der beiden Implementationen für fast alle getesteten Konfigurationen linear vom Anstieg von  $n_s$  abhängig. Erst bei einer Erhöhung von  $n_s$  von 150 auf 373 steigt die Laufzeit superlinear, der Anstieg ist aber weit von einem **sub-exponentiellen** Anstieg entfernt. Auf Basis der Ergebnisse wird vermutet, dass die PyFlink-Variante auch für Werte  $n_s > 373$  gut skaliert, während die Java-Variante aufgrund des superlinearen Wachstums der Laufzeit für eine noch größere Anzahl an Szenen möglicherweise schlecht skaliert.



(a) Durchschnittliche Laufzeit der PyFlink und Java-Flink-Implementationen für  $dop = 144$  in Abhängigkeit von  $n_s$ .



(b) Durchschnittliche Laufzeit der PyFlink und Java-Flink-Implementationen für  $n_s = 373$  in Abhängigkeit von  $dop$ .


Abbildung 4.2: Laufzeit der Flink-Implementationen für verschiedene Anzahlen von Szenen  $n_s$  und variierenden Grad der Parallelisierung  $dop$  mit  $dim_k = 500$  und  $dim_{tk} = 20$ .

#### Auswirkung des Grads der Parallelisierung $dop$ auf die Verarbeitungsgeschwindigkeit


Die in Abbildung 4.2b gezeigten Ergebnisse der Evaluierungen mit einem veränderten Grad der Parallelisierung für die beiden Flink-Varianten wirken zunächst widersprüchlich. Während die Java-Variante wie erwartet mit sinkendem  $dop$  eine erhöhte Laufzeit aufweist, sinkt diese für die PyFlink-Variante sogar ein wenig. Offenbar ist die Implementation des Schneide- und des Analysealgorithmus in Python im Vergleich zur Java-Variante enorm schnell, so dass die Zeitersparnis durch den wegen des geringeren  $dop$  gesunkenen Kommunikationsaufwand größer ist als die für die Verarbeitung der umfangreicheren Teilprobleme benötigte Zeit. Für noch kleinere Parallelisierungsgradwerte erhöht sich allerdings auch die Laufzeit der PyFlink-Version wie erwartet stark, ist der Java-Implementation aber zu jedem Zeitpunkt weit überlegen. Aus Zeitgründen konnten keine belastbaren Messungen für die Java-Variante mit  $dop < 12$  durchgeführt werden. Betrachtet man allerdings die Laufzeitsteigerung dieser Implementation bei sinkendem Grad der Parallelisierung ist anzunehmen, dass die Java-Variante auch bei einem  $dop = 6$  viel langsamer läuft als die PyFlink-Variante. Bei einer weiteren Testmessung, die für die PyFlink-Implementation mit  $dop = 1$  durchgeführt wurde, wurde eine Laufzeit von circa 15 Stunden ermittelt.

#### Skalierbarkeit der einzelnen Phasen des Algorithmus

Bei der Auswertung der beiden Evaluierung der beiden parallelisierten Varianten war die PyFlink-Variante für alle Konfigurationen schneller und besser skalierbar als das entsprechende Java-Flink Programm mit denselben Eingabeparametern. Um festzustellen, wie dieser extreme Anstieg der Verarbeitungsgeschwindigkeit möglich ist, wurden die Laufzeiten der einzelnen Phasen des Algorithmus untersucht. Die Flink-Implementationen bestehen dabei aus vier Phasen: dem Einlesen der Daten, dem Zerschneiden der Kacheln in Teilkacheln, der Analyse der Pixelzeitreihen sowie das

Schreiben der Daten in das verteilte Dateisystem. Aufgrund der parallelisierten Struktur des Algorithmus ist die Gesamtlaufzeit ungleich der Summe der Laufzeiten aller Phasen, da diese teilweise parallel verarbeitet werden. Da das Schreiben der Daten immer dann erfolgt, wenn die Analyse ein Ergebnis fertig berechnet hat, ist die Laufzeit hauptsächlich von der für die Analysephase benötigten Laufzeit abhängig und muss nicht weiter betrachtet werden. 

In den Abbildungen 4.3a und 4.3b sind Laufzeiten der einzelnen Phasen der PyFlink- bzw. der Java-Flink-Implementation bei einer Veränderung der Teilkachelkantenlänge  $dim_{tk}$  graphisch dargestellt. Während das Verhältnis der Laufzeiten der einzelnen Phasen in Abbildung 4.3a annähernd gleich bleibt, sind in Abbildung 4.3b deutliche Unterschiede zu erkennen. Die Wahl einer sehr kleinen Teilkachelkantenlänge  $dim_{tk} = 10$  führt zu einer deutlich zeitaufwändigeren Schneidephase, da die Anzahl der Teilkacheln  $n_{tk}$ , die aus den Kacheln herausgeschnitten werden müssen, dadurch deutlich steigt. In diesem Fall nimmt auch die Laufzeit der Analyse der Pixelzeitreihen zu. Vermutlich geschieht dies aufgrund der großen Anzahl an Teilkachelgruppen, die zudem sehr klein sind, so dass der Kommunikationsaufwand zwischen den Netzwerkknoten umfangreicher ist. Werden größere Teilkachelkantenlängen und somit weniger Teilkacheln genutzt, sinkt die Laufzeit und auch der Anteil dieser Phase an der Gesamtlaufzeit sukzessive. Im Gegensatz dazu steigt die Laufzeit der Analysephase bei größeren  $dim_{tk}$  an, da erheblich weniger Teilkacheln als mögliche Threads verarbeitet werden müssen. Im Extremfall  $dim_{tk} = 500$  ist diese Phase fast für die gesamte Laufzeit verantwortlich, da nicht geschnitten werden muss und die gesamte Analysephase in nur einem Thread ausgeführt werden muss.

Die stark beschleunigte Ausführungsgeschwindigkeit der Schneide- und Analysephase des PyFlink-Programms gegenüber den entsprechenden Phasen des Java-Flink-Programms lässt sich durch die Nutzung von effizient implementierten Datenstrukturen erklären. Beim Schneiden der Kacheln besteht die Hauptaufgabe des Schneide-Algorithmus aus dem Kopieren der relevanten Bereiche des Wertearrays der zu zerschneidenden Kachel, um das Wertearray der Teilkachel zu bilden. Während diese Wertearrays in Java als Arrays des Datentyps Short implementiert wurden, werden in Python ByteArrays verwendet. Diese verfügen über unterschiedlich implementierte Kopieroperationen. Die Kopierfunktion des Shortarrays legt ein neues Array an und kopiert dann die Werte in das neu erstellte Array. Letztendlich werden alle diese Arrays zu einem Array zusammengefügt. Das Kopieren des ByteArrays schreibt hingegen die Werte des relevanten Wertearraybereichs direkt in das bereits existierende ByteArray der Teilkachel. Die Analyse der Daten erfordert das Konstruieren der Pixelzeitreihen aus einer Teilkachelgruppe. Dies bedeutet, dass alle Pixelwerte der Wertearrays aller Teilkacheln der Gruppe einzeln gelesen und in eine neue Datenstruktur überführt werden müssen. Bei der Java-Flink-Variante werden die Pixelzeitreihen als HashMaps implementiert, in Python werden stattdessen Dictionarys verwendet. Dictionarys sind zwar ebenso wie HashMaps intern ebenfalls als Hashtable organisiert, beinhaltet aber im Gegensatz zu einer HashMap keine Objekte sondern bloß generische Datentypen, so dass das aufwändige extrahieren der Elemente aus den entsprechenden Objekten in der HashMap entfällt. Darüber hinaus ist möglicherweise die Analyse selbst in Python effizienter implementiert. 

### 4.3.2 Auswertung der Evaluierung der Python-Variante

Bei der Durchführung der Evaluierungen für die Python-Implementation wurden die Anzahl der Szenen  $n_s$  und die Kachelgröße, die den Zielbereich beschreibt,  $dim_k$  variiert. Dabei fiel auf, dass für alle 6 betrachteten Spektralbänder auf der genutzten Maschine keine Ergebnisse für die gesamte Testdatenmenge ermittelt werden konnten. Deshalb wurde  $n_s$  so gewählt, dass alle Bänder für die Analyse berücksichtigt werden konnten. Ein großes Problem war aber auch bei der Erhöhung von  $dim_k$  die Limitierung des verfügbaren Arbeitsspeichers auf 256 GB. Dies hat zur Ursache, dass der Python-Algorithmus die gesamte Datenmenge einliest und im Arbeitsspeicher hält während mit den Daten gearbeitet wird. Bei maximalem Wert  $n_s = 373$  war bereits das Einlesen von mehr



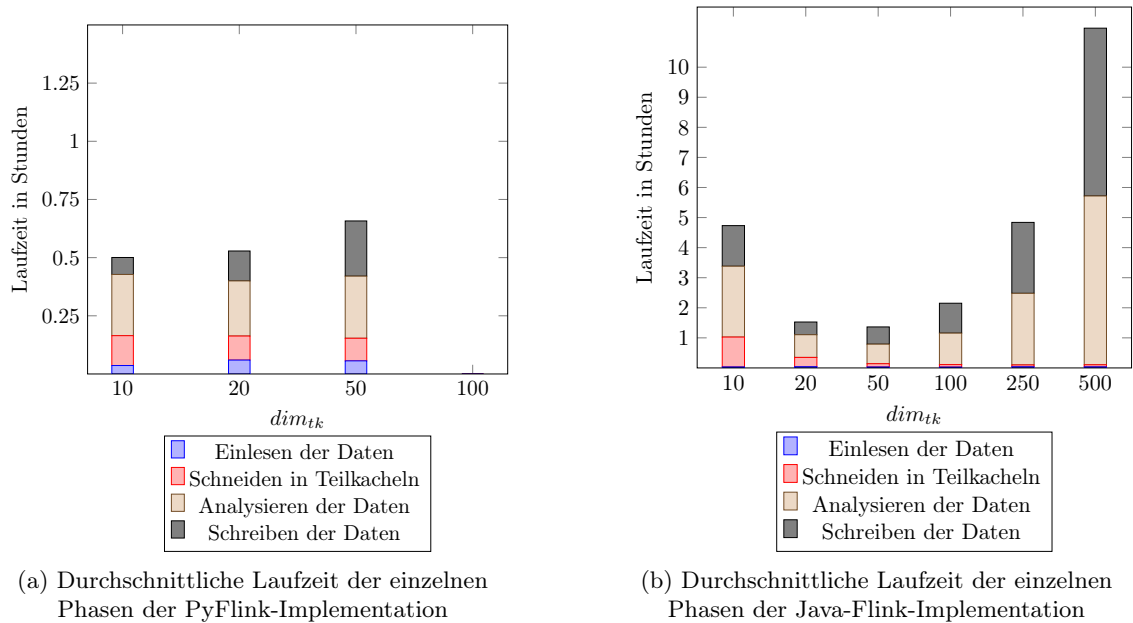


Abbildung 4.3: Graphische Darstellung der Skalierbarkeit der einzelnen Phasen der Flink-Implementation bei variablem Wert  $dim_k$

als zwei Bändern nicht mehr möglich, da nicht genügend Arbeitsspeicher zur Verfügung stand. Deshalb wurde der Effekt der Kachelkantenlänge  $dim_k$  auf die Laufzeit des Programms mit 150 Szenen mit jeweils 6 Spektralbändern untersucht.

#### Auswirkung der Kachelkantenlänge $dim_k$

Wie in Abbildung 4.4a zu erkennen ist, scheint die Kachelgröße in einem Bereich von  $50 \leq dim_k \leq 200$  so gut wie keine Auswirkung auf die Laufzeit zu haben. Erst bei  $dim_k = 400$  ist eine stark verlängerte Laufzeit festzustellen. Diese lässt sich anhand der Verteilung der Programmphasen „Einlesen der Daten“ und „Analysieren der Daten“ erklären. Während die benötigte Zeit zum Einlesen der Daten für alle Kachelkantenlängen auf einem ähnlichen Niveau befindet, verlängert sich die Zeit, die zum Analysieren der Pixelzeitreihen benötigt wurde, nicht linear mit zunehmender Kachelgröße. Grund dafür ist, ebenso wie bereits bei der Betrachtung der Auswirkung von  $dim_k$  auf die Laufzeit der parallelisierten Varianten, die Strukturierung der Daten, die dafür sorgt, dass bei kleineren Kacheln kaum valide Pixelwerte vorhanden sind. Mit zunehmender Kachelgröße sollte sich die Laufzeit weiter erhöhen, jedoch nicht so sprunghaft wie bei der Erhöhung von  $dim_k$  von 200 auf 400. Aufgrund der oben beschriebenen Limitierung des Arbeitsspeichers war es aber nicht möglich diese Versuchsreihe für größere  $dim_k$  fortzusetzen.

#### Auswirkung der Anzahl der Szenen $n_s$

Zusätzlich zur Kachelkantenlänge beeinflusst auch die Anzahl der zu verarbeitenden Szenen  $n_s$  die Laufzeit des Programms. Die Abbildung 4.4b zeigt die Laufzeiten der Python-Implementation bei einer variablen Szenenanzahl  $n_s$ . Dabei steigt die Laufzeit des Programms mit zunehmendem  $n_s$  fast linear. Dies widerspricht den Ergebnissen der Komplexitätsanalyse in [Unterabschnitt ??](#), die eine Komplexität von  $\mathcal{O}(n_b * dim_k^2 * n_s^4)$  ergibt. Jedoch bezieht sich sowohl die Abhängigkeit der



Laufzeit von  $dim_k$  als auch von  $n_s$  auf die Analyse, die, aufgrund der gewählten Kachelkantenlänge  $dim_k$  und der oben beschriebenen Menge an nicht validen Werten, jedoch nur in einigen Fällen durchgeführt wird. Da sich die gemessenen Laufzeiten zum größten Teil aus der linear von  $n_s$  abhängigen Einleseoperation ergeben, steigt die Gesamtlaufzeit des Programms nur etwas stärker als linear.

### Skalierbarkeit der einzelnen Phasen des Algorithmus

In beiden Diagrammen in Abbildung 4.4 ist klar ersichtlich, dass bereits das Einlesen der Daten einen erheblichen, meist sogar den größten, Teil der Laufzeit benötigt. Die Einlesegeschwindigkeit wird dabei entgegen der Annahme, dass die maximale Lesegeschwindigkeit der Festplatte das Einlesen verlangsamen würde, offenbar durch die in der Bibliothek GDAL implementierte Einlesefunktion der Szenen limitiert. Es ist jedoch keine Alternative ein eigenes Einleseverfahren zu entwickeln, da GDAL eine Standardbibliothek im Bereich der Geographie ist, die sehr populär ist. Die Analysephase ist aufgrund der angesprochenen Strukturierung der Eingabedaten zwar sehr schnell, dieser Wert ist aber wie bereits oben begründet nicht aussagekräftig. Erst bei der Nutzung einer Kachelkantenlänge von  $dim_k = 400$  nimmt die Analyse den Großteil der Laufzeit ein. Aufgrund der Limitierung des Arbeitsspeichers war es nicht möglich,  $dim_k > 400$  und  $n_s > 150$  zur selben Zeit zu wählen, so dass keine weiteren Evaluierungen durchgeführt werden konnten. Es wird aufgrund des für  $dim_k = 400$  bereits angedeuteten größeren Anteils der Analyse an der Gesamtlaufzeit eine schlechtere Skalierung erwartet.

Es wird also auf Basis der erhobenen Daten festgestellt, dass die Python-Implementation nicht gut skaliert. Da bereits eine Maschine mit 256 GB Arbeitsspeicher für die Laufzeittests verwendet wurde, kann es keine **nachhaltige** Alternative sein bei steigenden Datenmengen bzw. größeren zu analysierenden Zielgebieten Maschinen mit noch größerem Arbeitsspeicher zu verwenden. Stattdessen sollte der Fokus auf eine Optimierung des Quellcodes liegen. Während die Ausführungszeit für die Analyse akzeptabel, muss der Platzverbrauch von Arbeitsspeicher reduziert werden. Dies könnte zum Beispiel durch eine verbesserte Einlesefunktion erreicht werden, die nur noch die Daten des Zielgebiets in den Arbeitsspeicher lädt. Eine weitere Möglichkeit wäre es eine aufgrund der unzureichenden Hardwarekapazität nicht mehr ausführbare Analyse in mehrere Teilanalysen aufzuteilen. Diese könnten dann nacheinander durchgeführt werden, so dass nie mehr Arbeitsspeicher genutzt wird, als zur Verfügung steht. Dies würde genau dem parallelisierten Ansatz entsprechen, der mit Flink implementiert wurde.

### 4.3.3 Vergleich der parallelisierten und der nicht-parallelisierten Varianten

Nachdem die Skalierbarkeit und die Durchführungsgeschwindigkeit sowohl für die parallelisierten Varianten als auch für die Python-Implementation betrachtet wurde, sollen nun alle drei Varianten miteinander verglichen werden. Da die Python-Implementation lediglich die Parameter Anzahl der Szenen  $n_s$  und Kachelgröße  $dim_k$  verarbeiten kann, werden für alle drei Implementationen nur diese Eingabeparameter betrachtet. Für die parallelisierten Implementationen wird  $dop = 144$  und  $dim_{tk} = 20$  gesetzt, um eine gute Parallelisierbarkeit zu ermöglichen.

#### Vergleich der Auswirkungen der Anzahl der Szenen $n_s$

In Abbildung 4.5a ist gut zu erkennen, dass die Laufzeit der drei Varianten bei  $n_s = 20$  noch circa gleich ist. Die Laufzeit der Python-Implementation ist sogar marginal geringer als die der Java-Implementation. Dies ändert sich bei der Betrachtung der Ergebnisse für  $n_s = 50$ . Bereits bei dieser Anzahl an Szenen ist absehbar, dass die Python-Implementation schlechter skaliert als die

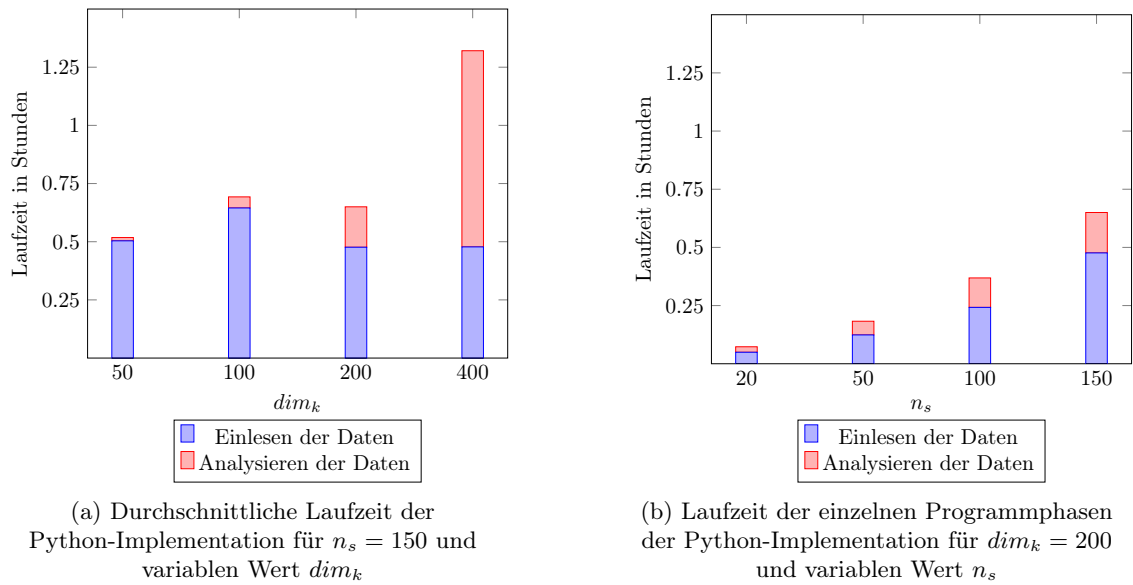
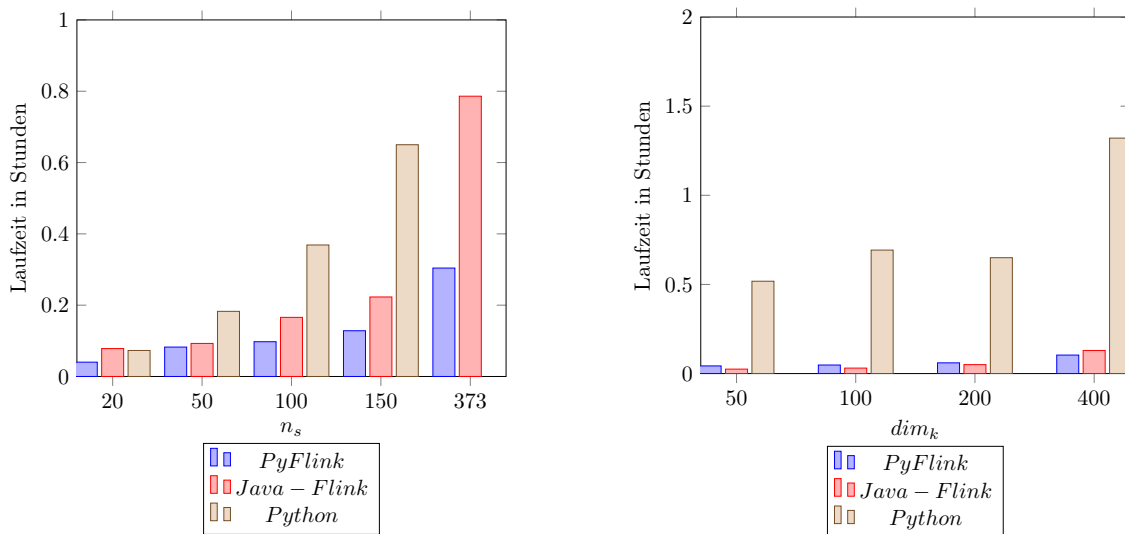


Abbildung 4.4: Graphische Darstellung der Skalierbarkeit der Python-Implementation.

parallelisierten Varianten, da sie bereits circa doppelt so viel Zeit benötigt wie die Java-Variante. Durch die Betrachtung der Ergebnisse für  $n_s = 100$  und  $n_s = 150$  bestätigt sich dieser Eindruck, da die Laufzeit der Python-Implementation weiterhin verhältnismäßig und absolut schneller ansteigt als die der parallelisierten Varianten. Für Werte  $n_s > 150$  konnten keine Ergebnisse für die Python-Implementation berechnet werden, da der Arbeitsspeicher der genutzten Maschine nicht ausgereicht hat.

Die Skalierbarkeit der Python-Implementation ist also nicht nur verhältnismäßig schlecht, sondern auch nur bis zu einer Anzahl an Szenen hin möglich, die bei der tatsächlichen Anwendung der Analyse auf ein geographisches Zielgebiet wohl leicht überschritten werden dürfte. Des weiteren ist  $dim_k = 200$  vergleichsweise gering gewählt, so dass für einen größeren Zielbereich die Anzahl der verarbeitbaren Szenen vermutlich noch geringer ist. Das bedeutet, dass bei der Nutzung der Python-Variante ein Kompromiss aus der Anzahl der zu analysierenden Szenen und der Größe des Zielbereichs gefunden werden muss, damit sie überhaupt angewendet werden kann. Dies wird bei der Nutzung der Flink-Varianten durch die parallelisierte Vorgehensweise sowie die Verteilung der zu verarbeitenden Daten auf die verschiedenen Maschinen durch Flink. So ermöglichte Flink die Berechnung einer Analyse auf einer Maschine, die bei der normalen Python-Variante an Arbeitsspeichermangel gescheitert wäre. **Verweis auf Anhang: Versuch mit dop = 1.** Dies ermöglicht eine gute Skalierbarkeit der Flink-Varianten für noch größere Datenmengen, auch wenn die Laufzeit dann vermutlich erheblich ansteigen würde.

Auch die Verarbeitungsgeschwindigkeit der Python-Variante ist durchweg geringer als die der parallelisierten Varianten. Dies liegt ebenfalls an dem soeben beschriebenen parallelen Einlesen der Daten aus dem verteilten Dateisystem in Flink, während Python alle Daten von einer Festplatte liest. Wie in Abschnitt 4.3.2 bereits beschrieben, ist die limitierte Lesegeschwindigkeit von der Festplatte jedoch nicht verantwortlich für die langsame Einlesegeschwindigkeit, sondern die nicht optimale Einleseoperation der genutzten Bibliothek GDAL.



(a) Durchschnittliche Laufzeit der Python-, der Java-Flink- und der Python-Flink-Implementationen für  $dim_k = 200$  in Abhängigkeit von  $n_s$ .

(b) Durchschnittliche Laufzeit der Python-, der Java-Flink- und der Python-Flink-Implementationen für  $n_s = 150$  in Abhängigkeit von  $dim_k$ .

Abbildung 4.5: Vergleich der Laufzeiten der Python-, der Java-Flink- und der Python-Flink-Implementationen bei Veränderung der Anzahl der Szenen  $n_s$  oder der Kachelkantenlänge  $dim_k$ .

### Vergleich der Auswirkungen der Kachelkantenlänge $dim_k$

Auch beim Vergleich der Auswirkungen der Variation der Kachelkantenlänge  $dim_k$  auf die Laufzeit der drei Varianten ist für alle evaluierten Werte klar ersichtlich, dass die Python-Variante die mit Abstand langsamste Implementationen ist.

### Vergleich der Stabilität

Während der Durchführung der Laufzeittests fiel auf, dass die unterschiedlichen Varianten nicht nur unterschiedlich gut skalieren beziehungsweise ausgeführt werden, sondern dass sie auch unterschiedlich stabil laufen. Obwohl es nicht vorgesehen war, wurde aufgrund der Häufung von Abstürzen einiger Implementationen beschlossen, dieses Kriterium ebenfalls zu untersuchen. In Tabelle 4.2 ist die kumulierte Anzahl der Abstürze der einzelnen Varianten aufgeführt. Außerdem wurde anhand der Gesamtzahl der Versuche für jede Variante die Anzahl von Abstürzen pro durchgeführtem Programmdurchlauf berechnet, so dass die Stabilität aller Varianten normiert wird und miteinander verglichen werden kann. Die mit Abstand stabilste Variante ist die Python-Implementation, die, falls die Arbeitsspeichergröße ausreichte, immer erfolgreich ausgeführt wurde. **Rest ergänzen**

Es wird erwartet, dass insbesondere die Stabilität von PyFlink mit fortschreitender Entwicklung des Systems verbessert werden wird. Da sich die Python-Programmierschnittstelle von Flink zum Zeitpunkt der Durchführung der Evaluation noch in der Beta-Phase befand, ist es auch nicht überraschend, dass das Verhältnis von Abstürzen zu Programmdurchläufen höher ist als für die Java-Programmierschnittstelle.

	Python	Java-Flink	PyFlink
Fehler	0		
Testdurchläufe	21		
Abstürze / Testdurchlauf	0		

Tabelle 4.2: Vergleich der Stabilität der drei Implementationsvarianten

# Kapitel 5

## Fazit

Ziel der vorliegenden Arbeit war die Implementierung und Evaluation eines Algorithmus zur Analyse von Pixelwerten von Satellitenbildern. Dieser schneidet aus einer gegebenen Szene einen Zielbereich aus. Dieser wird bei Bedarf nochmals in kleinere Teilkacheln zerteilt, die nach ihrer Position gruppiert werden. Für jede Pixelposition jeder Teilkachelgruppe wird dann eine Pixelzeitreihe konstruiert, die letztendlich mithilfe einer Support Vector Regression analysiert wird. Es wurden drei Varianten des Algorithmus implementiert. Dabei wurden die Java- sowie die Python-Programmierschnittstellen von Apache Flink, einem parallelisierten Datenverarbeitungssystem, sowie Python genutzt. Während die beiden Flink-Implementationen auf die Parallelisierung der Verarbeitungsprozesse mittels Flink ausgelegt sind, wurde die Python-Implementation sequentiell geplant und umgesetzt.

Um die drei Implementationen des Algorithmus evaluieren zu können, wurden die Evaluationskriterien Skalierbarkeit und Datenverarbeitungsgeschwindigkeit gewählt. Mithilfe eines Testdatensatzes von Satellitenbildern wurden verschiedene Testkonfigurationen durchgeführt, bei denen entweder die Anzahl der Szenen, die Größe des Zielbereichs, der Grad der Parallelisierung oder die Teilkachelgröße variiert wurde. Dabei wurden die letzten beiden Parameter lediglich für die parallelisierten Varianten berücksichtigt. **Es wurde angenommen, dass die nicht parallelisierte Variante sowohl schlechter skaliert als auch grundsätzlich eine geringere Verarbeitungsgeschwindigkeit aufweist.** Für die parallelisierten Varianten wurde aufgrund des Beta-Status von PyFlink angenommen, dass die Java-Flink-Variante besser skaliert und schneller ist als die PyFlink-Implementation. Während die erste Annahme durch die Evaluationen eindeutig bestätigt wurde, erwies sich die zweite Annahme als falsch. In den meisten Fällen war die PyFlink-Variante die mit Abstand schnellste und am besten skalierende Implementation. Lediglich die Stabilität von PyFlink und in gewissem Ausmaß auch Java-Flink ist ungenügend, da die Prozesse verhältnismäßig häufig aufgrund von Fehlern im Flink-System abgebrochen werden mussten.

Da sich Flink momentan noch in der Entwicklung befindet, ist damit zu rechnen, dass die Ergebnisse mit den kommenden Flink-Versionen noch besser werden und **das** insbesondere die Stabilität der Prozesse zu nimmt. Des weiteren ließe sich die Laufzeit reduzieren, wenn einzelne Teiloperationen optimiert implementiert werden würden. Dies gilt speziell für die Python-Variante, die eine verhältnismäßig langsame Einleseoperation einer Drittbibliothek nutzt. Außerdem könnten die Flink-Varianten erweitert werden, so dass die Analyse nicht nur die Regression der Werte sondern auch die Vorhersage zukünftiger Pixelwerte leistet. Eine weitere sinnvolle Erweiterung wäre das Speichern der Ergebnisse im Eingabedatenformat, damit die Ergebnisse besser visualisiert und weiter verarbeitet werden können. Interessant wäre auch der **Vergleich der Ergebnisse mit weiteren Evaluierungen von Flink.** Da Flink erst vor kurzer Zeit eingeführt wurde, fehlen jedoch belastbare Vergleichswerte. Insbesondere der Vergleich von Flink und einer sequentiellen Implementation

wurde noch nicht von anderen Autoren durchgeführt.

# Anhang A

## Testkonfigurationen der Evaluation des Algorithmus

Nachfolgend sind alle verwendeten Konfigurationen der Parameter Anzahl der Szenen  $n_s$ , Kantenlänge des Zielbereichs  $dim_k$ , Kantenlänge der Teilkacheln  $dim_{tk}$  und dem Grad der Parallelisierung.

### A.1 Konfigurationen der parallelisierten Varianten

#### A.1.1 Konfiguration für variable $n_s$

##### Versuch 1

$n_s = 20, dim_k = 500, dim_{tk} = 20, dop = 144$								
Phase	Java-Flink				PyFlink			
	1a	1b	1c	$\emptyset$	1a	1b	1c	$\emptyset$
Einlesen	00:07	00:07	00:04	00:06	00:26	00:08	00:05	00:13
Schneiden	03:40	02:53	03:00	03:11	01:01	00:46	01:06	00:58
Analyse	04:47	04:49	04:05	04:34	01:36	02:35	01:44	01:58
Schreiben	01:13	01:01	01:07	01:07	00:45	01:35	00:37	00:59
Gesamt	04:56	04:58	04:11	04:42	02:09	02:51	02:14	02:25

Tabelle A.1: Ergebnisse für  $n_s = 20$

##### Versuch 2

##### Versuch 3

#### A.1.2 Konfiguration für variable $dim_k$

### A.2 Zweiter Abschnitt

... oder hier.

# Literaturverzeichnis

- [AAP<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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] EMC<sup>2</sup>. The digital universe of opportunities. Technical report, EMC<sup>2</sup>, 2014.
- [Foua] Apache Software Foundation. Apache flink website. [https://ci.apache.org/projects/flink/flink-docs-release-0.9/internals/job\\_scheduling.html](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 apache<sup>TM</sup> flink<sup>TM</sup> as a top-level project. [https://blogs.apache.org/foundation/entry/the\\_apache\\_software\\_foundation\\_announces69](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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 29. Januar 2016

.....

## **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, 29th January 2016

.....