Masterarbeit

Evaluierung cloud-optimierter Datenformate zur Speicherung und Analyse großer raumzeitlicher Rasterdaten

Evaluation of Cloud-Optimized Data Formats for the Storage and Analysis of Large Spatiotemporal Raster Data

zur Erlangung des akademischen Grades Master of Science

13. Dezember 2024

Eingereicht von:	Christina Rathjen Jade Hochschule Oldenburg Fachbereich: Bauwesen Geoinformation Gesundheitstechnologie Studiengang: Geoinformationswissenschaften (Profil: Geoinformatik) Matrikelnummer: 6032301 christina.rathjen@student.jade-hs.de 4. Semester
1. Prüfer:	Prof. Dr. Thomas Brinkhoff Jade Hochschule Oldenburg
2. Prüfer:	Dr. Marcel Ziems Landesamt für Geoinformation und Landesvermessung Niedersachsen

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

AdV	Arbeitsgemeinschaft der Vermessungsverwaltungen der Bundesrepublik Deutschland
API	Anwendungsprogrammierschnittstelle
ARD	Analysis Ready Data
BSP	Bulk Synchronous Parallel
CEOS	Committee on Earth Observation Satellites
CNCF	Cloud Native Computing Foundation
COG	Cloud-Optimized Geotiff
CPU	Central Processing Unit
CRS	Koordinatenreferenzsystem
DBMS	Datenbankmanagementsystem
DFS	Distributed File Systems
DOP	Digitales Orthophoto
EO	Earth Observation
GB	Gigabyte
GIS	Geoinformationssystem
GPU	Graphics Processing Unit
HPC	High Performance Computers
IaaS	Infrastructure as a Service
IFD	Image File Directory
I/O	Input/Output
JSON	JavaScript Object Notation
LGLN	Landesamt für Geoinformation und Landesvermessung Niedersachsen
MB	Megabyte
MPI	Message Passing Interface
NDVI	Normalized Difference Vegetation Index
NIST	National Institute Of Standards And Technology
OGC	Open Geospatial Consortium
OSS	Object Storage Systems
PaaS	Platform as a Service
RAM	Random Access Memory
RDBMS	Relationales Datenbankmanagementsystem
SaaS	Software as a Service
STAC	Spatio-Temporal Asset Catalog
TB	Terrabyte
vCPU	Virtual Central Processing Unit
VM	Virtuelle Maschine

1 Einleitung

Die rasante Entwicklung der Erdbeobachtungstechnologien hat zu einer exponentiellen Zunahme raumzeitlicher Rasterdaten geführt, die zentrale Einblicke in Umweltprozesse, Urbanisierung und Klimaforschung ermöglichen. Mit immer fortschrittlicheren Technologien sind die Datenmengen von Terabytes auf Petabytes angewachsen und markieren das Zeitalter des „Big Data“ in der Geoinformation. Bereits 2014 generierten NASA-Missionen etwa 1,73 Gigabyte (GB) Erdbeobachtungsdaten pro Sekunde, wobei geplante Missionen zukünftig bis zu 24 Terrabyte (TB) pro Tag liefern sollen (Nicholas Skytland, 2014, o.S.). Der Großteil dieser Daten besteht aus Rasterdaten, die oft sowohl räumliche als auch zeitliche Dimensionen aufweisen. Solche raumzeitlichen Daten ermöglichen die Analyse komplexer räumlicher Muster und zeitlicher Veränderungen und liefern fundierte Grundlagen für Entscheidungen in Wissenschaft, Planung und Politik. Gleichzeitig wächst die Zahl der Nutzenden, die auf diese großen Datenmengen zugreifen möchten. Doch die enorme Menge und Komplexität stellen hohe Anforderungen an Speicherung, Verwaltung und Analyse.

Eine zentrale Herausforderung besteht dabei in der effektiven Erzeugung und Bereitstellung. Die Bundesregierung setzt auf Strategien wie das Open-Data-Gesetz und Initiativen der Geodateninfrastruktur Deutschland, um qualitativ hochwertige Geodaten trotz begrenzter Budgets effizient bereitzustellen (BMI, 2012, S. 7). Gleichzeitig stoßen herkömmliche Datenverwaltungssysteme an ihre Grenzen. Angesichts des exponentiellen Anstiegs an Datenmengen werden vermehrt Konzepte wie Parallelverarbeitung, verteilte Arbeitsweisen und Cloud Computing im Bereich der Geoinformation relevant. In diesem Zusammenhang haben sich cloud-native Technologien und cloud-optimierte Datenformate als Schlüsselelemente moderner Geoinformationssysteme etabliert. Sie bieten skalierbare Lösungen für die effiziente Speicherung und Analyse großer Datenmengen (Vögler et al., 2016, S.72). Optimierte Datenformate wie COG und Zarr sowie moderne Architekturen sind dabei entscheidende Bausteine.

Auch das Landesamt für Geoinformation und Landesvermessung Niedersachsen (LGLN) steht vor diesen Herausforderungen bei der öffentlichen Bereitstellung von Produkten wie digitalen Orthophotos und Höhenmodellen. Ein aktueller Trend in der Datenbereitstellung ist es, Rasterdaten in Form von COG mittels Spatio-Temporal Asset Catalog (STAC) bereitzustellen. Derzeit werden vom LGLN etwa 55.000 Digitale Orthophotos (DOPs) über STAC bereitgestellt. Dies umfasst für ca. zehn Jahre knapp 28 TB Rasterdaten. Ein alternativer Weg zur Bereitstellung raumzeitlicher Rasterdaten ist die Nutzung sogenannter Data Cubes. Diese zeigen insbesondere in der Fernerkundung ein wachsendes Interesse, da sie sofortige Analyse ohne aufwendige Vorverarbeitungsschritte erlauben. Datenformate wie Zarr bieten mit ihrer flexiblen und skalierbaren Struktur eine effiziente Lösung für mehrdimensionale Arrays wie Data Cubes, die auch im cloudbasierten Geodatenmanagement an Relevanz gewinnen.

Diese Arbeit zielt darauf ab, die cloud-optimierten Datenformate COG und Zarr für die Speicherung und Analyse großer raumzeitlicher Rasterdaten zu evaluieren. Durch die Analyse der Stärken und Grenzen der beiden Formate wird die Eignung für verschiedene Anwendungsfälle untersucht, insbesondere unter Berücksichtigung der zusätzlichen Herausforderungen, die sich durch die zeitliche Dimension ergeben. Dabei wird auch die Rolle moderner, cloudbasierter Verarbeitungstools und -architekturen untersucht, um die Nutzung solcher Formate zu optimieren und die wachsenden Anforderungen an Geodatenmanagementsysteme zu erfüllen. Die Evaluierung der Formate erfolgt anhand der DOPs des LGLN. Dadurch wird nicht nur die Eignung dieser Formate für raumzeitliche Analysen untersucht, sondern auch ein direktes Feedback zur aktuellen Datenbereitstellung des LGLN ermöglicht. Gleichzeitig werden Optimierungs- und Zukunftspotenziale aufgezeigt, insbesondere für die Integration der Zeitdimension in die Datenstrukturen des Geobasisdatenanbieters. Auf diese Weise bietet diese Arbeit Einblicke zu Lösungen für das cloudbasierte Geodatenmanagement von raumzeitlichen Rasterdaten und leistet einen Beitrag zur Optimierung der Speicherung und Analyse großer raumzeitlicher Rasterdaten, um den wachsenden Anforderungen und dem Streben nach einer benutzerfreundlichen Bereitstellung gerecht zu werden.

2 Theoretische Grundlagen

In diesem Kapitel der theoretischen Grundlagen werden zentrale Konzepte und Technologien wie Cloud Computing sowie parallele und verteilte Verarbeitung erläutert, einschließlich Dask, einem Framework für parallele und verteilte Datenverarbeitung in Python.

2.1 Cloud Computing

Die Verarbeitung raumzeitlicher Rasterdaten erfordert leistungsfähige Infrastrukturen, die nur wenige Organisationen besitzen. Cloud Computing bietet hier eine Lösung, indem es flexible und skalierbare IT-Ressourcen wie Rechenleistung und Speicher über das Internet bereitstellt (Yang et al., 2017, S. 120). Laut der Definition des National Institute Of Standards And Technology (NIST) ermöglicht Cloud Computing den Zugriff auf einen gemeinsamen Pool von Ressourcen, die von mehreren Nutzern über ein Netzwerk genutzt werden (Mell und Grance, 2011, S. 2). Zu den zentralen Merkmalen zählen die bedarfsgerechte Bereitstellung von Ressourcen, breiter Netzwerkzugang, schnelle Anpassungsfähigkeit und Mechanismen zur automatischen Optimierung. Neben den grundlegenden Merkmalen des Cloud Computing werden zumeist drei Cloud-Service-Modelle unterschieden. Diese umfassen Software as a Service (SaaS) (Bereitstellung von Anwendungen), Platform as a Service (PaaS) (Entwicklungsplattformen) und Infrastructure as a Service (IaaS) (Infrastruktur wie Speicher und Server). Darüber hinaus werden vier Bereitstellungsmodelle anhand des Nutzerkreises unterschieden: Private Cloud (exklusive Nutzung durch eine Organisation), Community Cloud (Nutzung durch Organisationen mit gemeinsamen Interessen), Public Cloud (offen für die Öffentlichkeit, bereitgestellt durch Drittanbieter) und Hybrid Cloud (Kombination verschiedener Modelle) (Mell und Grance, 2011, S. 2). Cloud Computing bietet somit zahlreiche Vorteile wie Kosteneffizienz, die hohe On-Demand-Verfügbarkeit, Skalierbarkeit und Unterstützung moderner Analysetechniken, birgt jedoch Herausforderungen wie Sicherheitsrisiken und Abhängigkeiten von Anbietern (Bengel et al., 2015, S. 472).

Der cloud-native Ansatz bezieht sich auf die Softwareentwicklung, bei der Anwendungen so konzipiert werden, dass sie die Vorteile der Cloud optimal nutzen. Laut der Cloud Native Computing Foundation (CNCF) ermöglicht der cloud-native Ansatz die Erstellung und Ausführung skalierbarer Anwendungen in modernen, dynamischen Umgebungen wie öffentlichen, privaten und hybriden Clouds (Cloud Native Computing Foundation, 2024, o.S.). Der cloud-native Ansatz, entstanden ab 2010, hat durch die rasante Entwicklung von Cloud-Technologien auch die Geowissenschaften stark beeinflusst. Mit zunehmendem Detailreichtum und Nutzwert von Geodaten wächst die Abhängigkeit von der Cloud, um Geoanwendungen und -dienste effizient und skalierbar zu erstellen und die stetig steigende Nachfrage zu bedienen (CyberSWIFT, 2022, o.S.). Der Fokus liegt dabei besonders auf der architektonischen Gestaltung, um die Eigenschaften der Cloud voll auszuschöpfen, anstatt lediglich monolithische Anwendungen in die Cloud zu portieren. Der Ansatz setzt auf Kernprinzipien wie Containerisierung, Microservices, unveränderliche Infrastrukturen und deklarative Anwendungsprogrammierschnittstellen (APIs), um Anwendungen widerstandsfähig und skalierbar zu machen. Die CNCF bietet mit der Trail Map einen Leitfaden zur Einführung cloud-nativer Technologien (Kristen Evans, 2018, o.S.). Typische Technologien basieren auf Containern, die Isolation und Portabilität ermöglichen, CI/CD für automatisierte Bereitstellung, Orchestrierung (z. B. Kubernetes) zur Verwaltung sowie Observability-Tools wie Prometheus für Monitoring. Cloud-native Anwendungen zeichnen sich durch Infrastrukturunabhängigkeit, Elastizität, Automatisierung und schnelle Wiederherstellung aus (Goniwada, 2022, S. 18f.). Dies ermöglicht die Entwicklung und den Betrieb von Geoanwendungen, die flexibel und skalierbar sind. Die Vorteile umfassen skalierbare Rechenleistung, hohe Verfügbarkeit, Portabilität und Kosteneffizienz, was den cloud-nativen Ansatz zu einer attraktiven Wahl macht.

2.2 Kubernetes

Kubernetes (oder kurz„K8s“) ist eine Open-Source-Plattform zur Orchestrierung von Containern, die von Google entwickelt wurde und auf der internen Plattform Borg basiert, die Google viele Jahre in seinen

Rechenzentren nutzte. 2015 wurde Kubernetes Teil der Cloud Native Computing Foundation (Goniwada, 2022, S. 600). Seitdem ist Kubernetes der de facto Standard im Bereich Container und Orchestrierung (FreeWheel Biz-UI Team, 2024, S. 14). Kubernetes bietet zahlreiche fortschrittliche Funktionen, wie Sicherheit, Monitoring und Skalierung, die den Betrieb von containerisierten Anwendungen effizienter und robuster machen (The Kubernetes Authors, 2024b, o.S.). Es ermöglicht eine automatische Skalierung der Anwendungen, sowohl horizontal (Hinzufügen oder Entfernen von Containern) als auch vertikal (effiziente Ressourcenzuweisung), je nach aktueller Auslastung. Kubernetes gewährleistet zudem eine hohe Verfügbarkeit durch Selbstheilung, indem ausgefallene Container automatisch neu gestartet werden (Khushalani, 2022, S.8). Darüber hinaus ermöglicht Kubernetes automatisierte Rollouts und Rollbacks, sodass neue Versionen schrittweise eingeführt und im Falle von Fehlern problemlos auf frühere Versionen zurückgesetzt werden können. Ein weiterer wesentlicher Vorteil ist die Flexibilität und Portabilität von Kubernetes. Es unterstützt Multi-Cloud-Strategien und kann in verschiedenen Umgebungen wie Public Cloud, Private Cloud, Hybrid-Cloud oder On-Premises betrieben werden, was die einfache Migration von Anwendungen sowie eine hohe Ausfallsicherheit ermöglicht (The Kubernetes Authors, 2024b, o.S.).

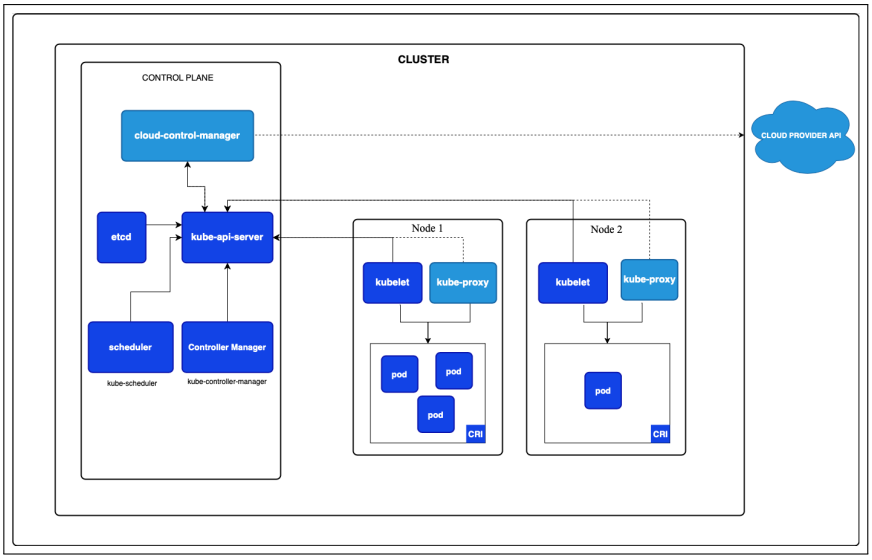

Abbildung 1: Kubernetes-Architektur (The Kubernetes Authors, 2024a, o.S.)

Kubernetes ist ein System zur Automatisierung der Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen. Es verwendet eine Cluster-Architektur, die aus Master- und Node-Komponenten besteht. Um die Funktionsweise zu erklären, ist zunächst die Betrachtung der Komponenten aus Abbildung 1 sinnvoll (Khushalani, 2022, S. 12f.):

- **Master:** Der Master ist die zentrale Steuerung des Clusters. Dieser Knoten ist ganz links zu sehen und bildet das sogenannte Control Plane. Seine Hauptaufgaben umfassen die Verwaltung von Anfragen, die Koordination der Ressourcen und das Speichern von Konfigurationen. Der Master besteht aus mehreren Komponenten (Khushalani, 2022, S. 12f.):
  - **API-Server:** Dies ist der Eingangspunkt für alle Anfragen an den Cluster. Jede Kommunikation oder Aktion, die auf den Cluster zugreifen will, geht über diesen REST-basierten Server.
  - **Scheduler:** Sobald eine neue Aufgabe im Cluster ankommt, entscheidet der Scheduler, auf welchem Arbeitsknoten (Node) die Aufgabe ausgeführt werden soll. Er wählt basierend auf den verfügbaren Ressourcen wie Central Processing Unit (CPU), Random Access Memory (RAM) und Speicher den besten Knoten aus.
  - **Controller:** Der Controller überwacht kontinuierlich den Zustand des Clusters. Er sorgt dafür, dass alle Komponenten wie gewünscht arbeiten und startet Prozesse neu, wenn sie fehlschlagen.

  - **Cluster-Speicher (etcd):** Hier werden alle Konfigurationen und Informationen über den Zustand des Clusters gespeichert. Etcd ist ein verteilter Schlüssel-Werte-Speicher, den Kubernetes nutzt, um sicherzustellen, dass alle Daten über den Cluster jederzeit zugänglich sind.

- **Node:** Ein Node in Kubernetes ist ein Server, der mit dem Master (Steuerebene) verbunden ist und die eigentliche Arbeit im Cluster ausführt. Jeder Node hat seine eigenen Hardware- und Betriebssystem-Spezifikationen, was bedeutet, dass verschiedene Nodes im Cluster unterschiedliche Betriebssysteme wie Windows oder Linux haben können. Die Hauptkomponenten eines Nodes sind:

  - **Kubelet:** Kubelet sorgt dafür, dass die Container-Anwendungen auf dem Node gemäß den Anweisungen des Masters korrekt ausgeführt und überwacht werden. Wenn der Master entscheidet, dass ein neuer Container auf diesem Node erstellt werden soll, übernimmt Kubelet die Koordination (Khushalani, 2022, S. 13).
  - **Container Runtime Interface (CRI)**: Kubernetes nutzt CRI, um Container zu starten, zu verwalten und zu überwachen. Sobald Kubelet den Befehl erhält, einen Container zu erstellen, leitet es diese Aufgabe an die CRI weiter, die den Container auf dem Node startet (Khushalani, 2022, S. 13).
  - **Pods:** Die Workloads werden dann in Form von Pods ausgeführt. Pods sind die kleinste deploybare Einheit und enthalten einen oder mehrere Container, die die Anwendungen ausführen. Ein Pod hat eine IP-Adresse, einen Namen und einen Portbereich, die von allen Containern innerhalb des Pods gemeinsam genutzt werden. Pods sind kurzlebig und können jederzeit entfernt, hinzugefügt oder verschoben werden. Die Container in einem Pod interagieren miteinander über das Dateisystem oder das Netzwerk. Dieser Netzwerkverkehr wird über den Kube-proxy verwaltet (Goniwada, 2022, S. 602f.).

Zusätzlich gibt es Mechanismen wie ConfigMaps und Secrets zur Verwaltung von Konfigurationsdaten und Passwörtern sowie Jobs für einmalige oder zeitgesteuerte Aufgaben (Khushalani, 2022, S. 49f.). Darüber hinaus umfasst das Kubernetes-Ökosystem eine Vielzahl von Werkzeugen, die die Funktionalität der Plattform erweitern. Die Integration von Monitoring- und Logging-Lösungen wie Prometheus ermöglichen beispielsweise die Überwachung und das Log-Management. Helm ist ein weit verbreiteter Paketmanager für Kubernetes, der die Verwaltung und Bereitstellung von Anwendungen erheblich vereinfacht (Khushalani, 2022, S. 33). Ein Helm Chart enthält alle notwendigen Kubernetes-Ressourcen, die für die Installation einer Anwendung erforderlich sind, einschließlich Deployments, Services und ConfigMaps. Um eine Anwendung mit Helm zu installieren, wird zuerst ein Kubernetes-Cluster über die Weboberfläche eines Cloud-Dienstes oder CLI-Tools erstellt. Nach der Cluster-Erstellung wird es mit *kubectl* verwaltet. Anschließend wird ein Helm-Chart heruntergeladen und die *values.yaml*-Datei angepasst. Diese Konfiguration beschreibt die gewünschten Ressourcen wie Deployments, Pods und Services. Mit dem Befehl *helm install* <*release-name*> <*chart*> wird die Anwendung bereitgestellt. Helm rendert diese Datei basierend auf den Werten in der values.yaml-Datei und sendet sie dann an den Kubernetes-API-Server. Der API-Server sorgt dafür, dass die notwendigen Ressourcen erstellt werden, z. B. Pods, die die Anwendung ausführen und Services, die den Netzwerkzugriff zu den Pods ermöglichen. Änderungen an der Anwendung oder deren Konfiguration können durch Aktualisieren der values.yaml und Ausführen von *helm upgrade* <*release-name*> <*chart*> vorgenommen werden.

2.3 Grundkonzepte der parallelen und verteilten Verarbeitung

Parallele und verteilte Verarbeitung sind fundamentale Konzepte in der modernen Datenverarbeitung, insbesondere im Kontext von Big Data und Cloud Computing. Das Hauptziel besteht darin, die Ausführungsgeschwindigkeit von Anwendungen zu steigern, die auf Multiprozessorsystemen, High Performance Computers (HPC) oder in der Cloud ablaufen (Bengel et al., 2015, S. 23). Diese Ansätze ermöglichen es, komplexe Berechnungen und Datenanalysen effizienter durchzuführen, indem Aufgaben auf mehrere Recheneinheiten verteilt werden. Dazu werden in diesem Kapitel zunächst die theoretischen Grundlagen

und Begriffe im Kontext der Speicherung und Analyse raumzeitlicher Rasterdaten erläutert. Der zweite Abschnitt widmet sich der Performance paralleler und verteilter Verarbeitung und zeigt die Grenzen der Laufzeitverbesserung auf. Kapitel 2.3.3 konzentriert sich schließlich auf die praktische Anwendung der Konzepte von Parallelität und verteilter Verarbeitung und stellt in diesem Zusammenhang verschiedene Programmiermodelle vor.

2.3.1 Parallele und verteilte Verarbeitung

Die **parallele Verarbeitung** bezieht sich auf die gleichzeitige Ausführung mehrerer Berechnungen innerhalb eines einzelnen Computersystems. Dabei werden Aufgaben auf die verschiedenen Prozessoren oder Prozessorkerne des Computers verteilt, sodass mehrere Aufgaben gleichzeitig bearbeitet werden können. Im Kontext der Verarbeitung von Rasterbildern bedeutet dies, dass mehrere Raster gleichzeitig auf demselben Computer verarbeitet werden können. Eine Anwendung oder Berechnung kann dabei in kleinere, unabhängige Einheiten zerlegt werden, sogenannte Tasks (Kuzmiakova, 2022, S. 10). Tasks sind die abstrakte Repräsentation einer Einheit. Auf Betriebssystemebene können Tasks als Prozesse oder Threads implementiert werden. Ein Prozess ist ein eigenständiges Programm mit eigenem Speicherbereich, wobei ein Prozess mehrere Threads enthalten kann, die parallel arbeiten und sich denselben Speicher teilen.

Die **verteilte Verarbeitung** erweitert dieses Konzept, indem sie mehrere unabhängige Computersysteme oder Knoten vernetzt, die zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen (Kuzmiakova, 2022, S. 144). Jeder Knoten ist ein eigenständiger Computer (physisch oder virtuell), der autonom arbeitet und mit den anderen Knoten kommuniziert, um Daten auszutauschen und Aufgaben zu koordinieren. Dies ermöglicht die Skalierung über die Leistungsgrenzen eines einzelnen Systems hinaus, indem zusätzliche Knoten hinzugefügt werden. Rasterdaten werden in diesem Fall auf verschiedene Knoten verteilt und jeder Knoten verarbeitet seinen Teil der Rasterbilder unabhängig von den anderen. Dadurch können viele Rasterbilder gleichzeitig verarbeitet werden, was die Gesamtverarbeitungszeit verringert. Allerdings gehen damit höhere Kommunikations- und Koordinationskosten einher.

Nebenläufigkeit und asynchrone Verarbeitung:
Dabei ist es wichtig, Parallelität und Nebenläufigkeit (Concurrency) nicht zu verwechseln. Nebenläufigkeit bedeutet, dass mehrere Aufgaben gleichzeitig organisiert werden, sodass sie unabhängig voneinander voranschreiten können. Das bedeutet, dass Aufgaben sich zeitlich überlappen können, auch wenn sie nicht gleichzeitig auf unterschiedlichen Prozessoren bzw. Computersystemen laufen. In einem nebenläufigen System können also Aufgaben zeitlich verschränkt (quasi-parallel) ausgeführt werden, indem das System zwischen ihnen wechselt. Parallelität meint dagegen die tatsächliche gleichzeitige Ausführung mehrerer Aufgaben auf verschiedenen Prozessoren bzw. CPUs (Kuzmiakova, 2022, S. 4f.).

In vielen parallelen und verteilten Systemen wird die asynchrone Verarbeitung verwendet, um Effizienzgewinne zu erzielen. Dabei werden Aufgaben nicht sequenziell, sondern unabhängig voneinander bearbeitet. Asynchrone Verarbeitung ist eine Technik, die es ermöglicht, dass Aufgaben unabhängig voneinander gestartet und beendet werden. Dabei müssen Tasks nicht aufeinander warten, sondern können parallel ablaufen. Asynchronität ist besonders nützlich, wenn es Aufgaben gibt, die auf das Ergebnis anderer Aufgaben warten müssen, z. B. bei Input/Output (I/O)-gebundenen Aufgaben. So können Aufgaben, wie das Lesen und Schreiben von Rasterdateien, effizienter ablaufen, da das System während der Wartezeit an anderen Aufgaben arbeiten kann. Beim Laden von Rasterbildern in einem verteilten System können diese asynchron geladen werden, während bereits verarbeitete Daten gleichzeitig gespeichert werden. Das System wartet nicht darauf, dass der Ladevorgang vollständig abgeschlossen ist, sondern fährt mit der Verarbeitung anderer Daten fort.

Arten paralleler Aufgaben:
In der parallelen Verarbeitung wird primär zwischen I/O-gebundenen und CPU-gebundenen Aufgaben unterschieden:

- **I/O-gebundene Aufgaben:** Diese sind durch hohe Wartezeiten bei Ein- und Ausgabeoperationen charakterisiert, wie beispielsweise das Lesen und Schreiben von Rasterdateien. Diese Prozesse profitieren von paralleler Verarbeitung, da mehrere I/O-Operationen gleichzeitig ausgeführt werden können, wodurch die Wartezeiten reduziert werden.

- **CPU-gebundene Aufgaben:** Solche Aufgaben werden vorwiegend durch die Rechenleistung des Prozessors bestimmt. Beispiele hierfür sind die Berechnung von räumlichen Indexen, Statistiken und Trendanalysen über die Zeit. Diese Aufgaben erfordern intensive numerische Berechnungen und profitieren von der Verteilung der Rechenlast auf mehrere Prozessorkerne.

Parallelisierungsstrategien:
Um die Parallelität umzusetzen, gibt es zwei Hauptmechanismen, die je nach Anwendungsfall und Systemarchitektur eingesetzt werden:

- **Datenparallelismus**: Bei dieser Form werden große Datensätze in kleinere Partitionen aufgeteilt, die parallel verarbeitet werden können. Jeder Prozessor oder Knoten führt die gleiche Operation auf einem unterschiedlichen Teil des Datensatzes aus. Dies ist besonders effektiv bei rechenintensiven Aufgaben mit hohem Datenvolumen. In der Rasterdatenverarbeitung könnten verschiedene Bilder oder Bildbereiche auf unterschiedliche Prozessoren oder Knoten verteilt werden, die jeweils parallel die gleichen Verarbeitungsschritte durchführen.

- **Aufgabenparallelismus (Task-Parallelismus)**: Hierbei werden verschiedene Aufgaben parallel auf verschiedenen Prozessoren ausgeführt. Unterschiedliche Prozessoren, Kerne oder Knoten bearbeiten unterschiedliche Aufgaben, die unabhängig voneinander sind oder miteinander interagieren. Beispielsweise könnte ein Prozessor Rasterbilder laden, während ein anderer sie verarbeitet oder analysiert.

Multithreading vs. Multiprocessing:
Um diese parallele Verarbeitung auf einem einzelnen Rechner durchzuführen, können entweder Multithreading oder Multiprocessing verwendet werden:

- **Multithreading:** Beim Multithreading werden mehrere Threads innerhalb eines Prozesses parallel ausgeführt. Die Threads teilen sich denselben Speicher, was die Kommunikation zwischen ihnen erleichtert. Dies ist besonders effektiv, wenn die Aufgaben hauptsächlich I/O-gebunden sind.

- **Multiprocessing:** Beim Multiprocessing werden viele Prozesse gleichzeitig ausgeführt. Diese arbeiten unabhängig voneinander und verfügen jeweils über ihre eigenen Speicherbereiche. Dies eignet sich tendenziell für CPU-gebundene Aufgaben, da jeder Prozess einen eigenen CPU-Kern nutzen kann.

Skalierbarkeit:
Nehmen die zu verarbeitenden Rasterdatenmengen zu, erweitert die verteilte Verarbeitung dieses Konzept, indem mehrere Knoten vernetzt werden. Dies ermöglicht die Skalierung über die Grenzen eines einzelnen Systems hinaus. Ein zentrales Konzept ist an dieser Stelle die Skalierbarkeit. Skalierbarkeit beschreibt die Fähigkeit eines Systems, seine Leistungsfähigkeit an veränderte Anforderungen anzupassen. Dies ist wichtig, um einerseits bei hoher Arbeitsbelastung zuverlässig und leistungsstark zu bleiben und andererseits bei geringerer Auslastung weniger Ressourcen zu verbrauchen. Dabei wird grundsätzlich zwischen vertikaler und horizontaler Skalierbarkeit unterschieden (Moreno et al., 2022, S. 5):

- **Horizontale Skalierung (Scale-out):** Dies meint das Hinzufügen oder Entfernen weiterer Knoten. Die Ressourcen der einzelnen Knoteninstanzen bleiben unverändert.

- **Vertikale Skalierung (Scale-up):** Hierbei bleibt die Anzahl der Knoten konstant, aber die Ressourcen werden z. B. durch mehr CPU-Kerne oder mehr Speicher erhöht oder verringert.

Aufgabenplanung und Lastverteilung:
In komplexen Systemen ist das Task-Scheduling von entscheidender Bedeutung. Task-Scheduling bestimmt, welche Aufgaben wann und wo ausgeführt werden, um Ressourcen optimal zu nutzen. Die Tasks werden auf mehrere Prozessoren oder Knoten verteilt und parallel ausgeführt. Die Verteilung nimmt ein Ablaufplaner, auch genannt Scheduler, vor (Bengel et al., 2015, S. 23). Zum Beispiel könnte der Scheduler festlegen, dass Prozessor 1 die ersten 1.000 Rasterbilder analysiert, während Prozessor 2 die nächsten 1.000 bearbeitet. Bei verteilten Systemen würde der Scheduler die Bilder entsprechend auf die verschiedenen Knoten verteilen. Die Granularität bestimmt die Anzahl und Größe der Tasks, die vom Scheduler verwaltet werden müssen (Singh, 2021, S. 12). Eine feine Granularität führt zu vielen kleinen Tasks (z. B. pro Bild), die effizient verwaltet werden müssen, um Overhead zu minimieren. Eine grobe Granularität dagegen reduziert die Anzahl der Tasks und den Overhead, kann aber zu Lastungleichgewichten führen. Ein effizienter Scheduler sorgt dafür, dass die Tasks gleichmäßig verteilt werden und keine Ressourcen überlastet werden. Dies wird auch als Load Balancing bezeichnet. In verteilten Systemen müssen dabei auch Faktoren wie Netzwerkbandbreite und Kommunikationskosten berücksichtigt werden. Ein effizientes Scheduling berücksichtigt die Abhängigkeiten zwischen Aufgaben, die verfügbaren Ressourcen und strebt eine Minimierung des Gesamtoverheads an.

Herausforderungen:
Overhead meint den zusätzlichen Aufwand durch die Verwaltung und Koordination in parallelen und verteilten Systemen. Es gibt zwei Arten von Overhead:

- **Kommunikations-Overhead:** Dies bezeichnet die Zeit, die für den Datenaustausch zwischen Prozessoren oder Knoten benötigt wird.
- **Synchronisations-Overhead:** Dieser meint den Aufwand für die Koordination des Zugriffs auf gemeinsame Ressourcen, was zu Wartezeiten und Ineffizienzen führt.

Der Overhead kann dazu führen, dass die Effizienz ab einer bestimmten Anzahl von Ressourcen sinkt, da der Aufwand für die Synchronisation den Nutzen der zusätzlichen Ressourcen überwiegt. Weitere typische Probleme bei der parallelen und verteilten Verarbeitung sind (Palach, 2014, S. 13f.):

- **Deadlocks:** Dabei blockieren sich Prozesse gegenseitig, weil jeder auf Ressourcen wartet, die von einem anderen gehalten werden.
- **Starvation:** Ein Prozess erhält nie die benötigten Ressourcen, weil andere Prozesse mit höherer Priorität bevorzugt werden.
- **Race Conditions:** Diese können auftreten, wenn das Ergebnis eines Prozesses von der Reihenfolge des Zugriffs auf gemeinsam genutzte Daten abhängt.

Diese Probleme können durch den Einsatz von Synchronisationsmechanismen wie Locks, Semaphoren und Barrieren gemindert werden, die den Zugriff auf Ressourcen koordinieren und sicherstellen, dass Aufgaben in der richtigen Reihenfolge ausgeführt werden (Bengel et al., 2015, S. 62). In verteilten Systemen ist auch die Fault Tolerance entscheidend. Dabei wird gewährleistet, dass einzelne Knoten ausfallen können, ohne die gesamte Verarbeitung zu beeinträchtigen. Durch Mechanismen wie Datenreplikation und Wiederherstellung (Recovery) können Aufgaben fortgesetzt und Ausfälle kompensiert werden. Dies erhöht Zuverlässigkeit und verhindert Totalausfälle (Bengel et al., 2015, S. 26).

2.3.2 Leistungsmetriken

Das Hauptziel der parallelen und verteilten Verarbeitung besteht darin, die Laufzeit von Programmen zu verkürzen. Dabei beschreibt die Laufzeit den Zeitraum zwischen dem Start eines parallelen Programms und der vollständigen Abarbeitung aller beteiligten Prozesse. Beeinflusst wird die Laufzeit durch Faktoren wie die Art der verwendeten Hardware, der Grad der Parallelität und das verwendete parallele Programmiermodell. Um die Leistung eines parallelen Algorithmus bewerten zu können, sind geeignete Leistungsmetriken erforderlich. Im Folgenden werden wichtige Metriken zur Leistungsbewertung, wie

Speedup und *Effizienz*, erläutert. Außerdem wird auf die durch das Amdahl'sche Gesetz definierten Grenzen der Parallelisierung eingegangen.

Um die Leistungssteigerung durch parallele Verarbeitung zu bewerten, wird der Speedup (S) bestimmt. Speedup S ist definiert als das Verhältnis der Ausführungszeit eines sequentiellen Programms zur Ausführungszeit des parallelen Programms auf einer bestimmten Anzahl N an Prozessoren (Kuzmiakova, 2022, S. 156). Dieser Wert zeigt, wie viel schneller eine Aufgabe durch den Einsatz mehrerer Prozessoren ausgeführt wird. Das Ziel ist, den Speedup zu maximieren, um die Effizienz der parallelen Verarbeitung zu steigern. Es gibt drei Arten von Skalierung, die den Effekt des Speedups beschreiben (Bengel et al., 2015, S. 341):

- **Sublineare Skalierung:** Die Leistungssteigerung ist geringer als die Zunahme der Ressourcen, oft aufgrund von Overhead oder Ressourcenkonflikten.
- **Lineare Skalierung:** Die Leistung steigt proportional zur Zunahme der Ressourcen.
- **Superlineare Skalierung:** Die Leistungssteigerung ist größer als die Zunahme der Ressourcen, was selten vorkommt und oft auf Caching-Effekte zurückzuführen ist.

Abbildung 2 veranschaulicht diese drei Arten von Speedup. Während die lineare Skalierung der Idealzustand ist, zeigt der reale Speedup, dass in der Praxis meist nur eine sublineare Skalierung erreicht wird.

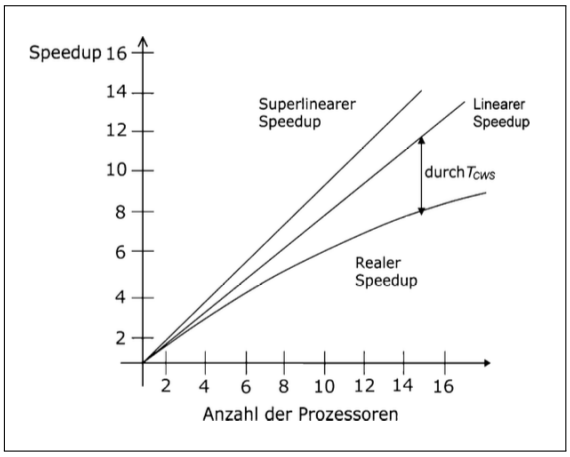

Abbildung 2: Sublinearer, linearer und superlinearer Speedup (Bengel et al., 2015, S. 341)

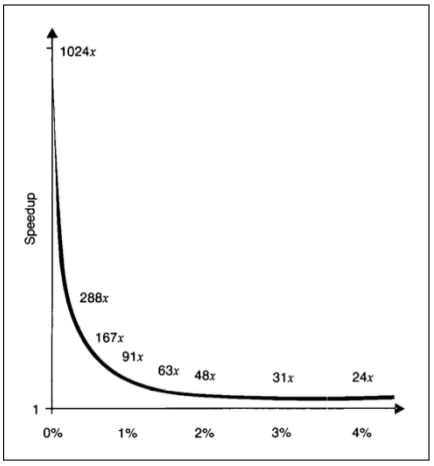

Abbildung 3: Speedup beim Amdahl´schen Gesetz (Gustafson, 1988, S. 532)

Die Effizienz E bewertet, wie gut die Prozessoren genutzt werden und ist definiert durch $E = S/N$, wobei N die Anzahl der Prozessoren und S der Speedup ist (Bengel et al., 2015, S. 342). Eine Effizienz von 1 bedeutet eine perfekte Skalierung, was in der Praxis selten erreicht wird.

Amdahls Gesetz:
Eine der grundlegendsten theoretischen Begrenzungen der Parallelisierung wird durch das Amdahl'sche Gesetz beschrieben. Es besagt, dass die maximal erreichbare Beschleunigung eines Programms durch den Anteil der Aufgabe begrenzt ist, der nicht parallelisiert werden kann (Kuzmiakova, 2022, S. 128f.). Dies bedeutet, dass selbst bei einer unbegrenzten Anzahl an Prozessoren die Beschleunigung eines Programms durch serielle Teile des Codes eingeschränkt bleibt. Dies wird mathematisch beschrieben durch:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Dabei ist S(N) der maximale Speedup mit N Prozessoren, P der Anteil des Programms, der parallelisiert werden kann (zwischen 0 und 1) und (1-P) der sequentielle Anteil, der nicht parallelisiert werden

kann. Abbildung 3 zeigt den Zusammenhang zwischen dem Speedup und dem nicht-parallelisierbaren Anteil eines Programms im Kontext des Amdahl'schen Gesetzes. Dies ähnelt mathematisch dem Verlauf einer hyperbolischen Funktion. Bei einem nahezu vollständig parallelisierbaren Programm (nahe 0% nicht-parallelisierbarer Anteil) kann der Speedup drastisch ansteigen. Bereits ein geringer Zuwachs des nicht-parallelisierbaren Anteils führt jedoch zu einer deutlichen Abnahme des maximal möglichen Speedups. Je größer der nicht-parallelisierbare Anteil eines Programms ist, desto geringer ist der zusätzliche Nutzen von mehr Prozessoren. Bereits ein kleiner serieller Anteil (z. B. 1-2%) kann den maximal erreichbaren Speedup einschränken. Das bedeutet, dass zum Beispiel bei einem Programm, bei dem 90 Prozent des Codes parallelisiert werden können, aber 10 Prozent seriell bleiben müssen, die maximal erreichbare Beschleunigung selbst bei einer unendlichen Anzahl von Prozessoren nur den Faktor 10 beträgt (Bengel et al., 2015, S. 343). Eng im Zusammenhang mit dem Amdahl´schen Gesetz steht das Prinzip des abnehmenden Grenznutzens (Diminishing Returns). Das Prinzip des abnehmenden Grenznutzens besagt, dass jeder zusätzliche Prozessor weniger zur Gesamtbeschleunigung beiträgt als der vorherige. Dies ist eine direkte Folge des Amdahl'schen Gesetzes und wird durch die zunehmenden Overheads bei der Koordination und Kommunikation zwischen den Prozessoren verstärkt.

Gustafsons Gesetz:
Das Amdahl'sche Gesetz beschreibt den maximal erreichbaren Speedup bei konstanter Problemgröße. Im Gegensatz dazu nimmt das Gustafson'sche Gesetz an, dass die Problemgröße mit der Anzahl der Prozessoren wächst. Gustafson argumentiert, dass in der Praxis der parallelisierbare Anteil des Problems mit der Anzahl der Prozessoren und der Datenmenge zunimmt, was zu einem nahezu linearen Speedup führt. Das bedeutet, dass bei zunehmender Prozessoranzahl auch größere Probleme effizienter gelöst werden können, da der sequentielle Anteil relativ kleiner wird. Dadurch kann bei großen Systemen ein Speedup nahezu proportional zur Anzahl der Prozessoren erreicht werden (Bengel et al., 2015, S. 344).
Eine Anwendung gilt als skalierbar, wenn die Laufzeit bei wachsender Problemgröße und steigender Prozessoranzahl konstant bleibt. In diesem Fall bleibt auch die Effizienz des parallelen Programms konstant. Welches Gesetz zutrifft, hängt von der Skalierbarkeit der Anwendung ab (Bengel et al., 2015, S. 345). Das Amdahl'sche Gesetz gilt für nicht skalierbare Anwendungen. Das Gustafson'sche Gesetz gilt für perfekt skalierbare Anwendungen.

2.3.3 Architekturtypen und Programmiermodelle

Die erfolgreiche Umsetzung paralleler und verteilter Verarbeitung hängt maßgeblich von der Wahl der richtigen Hardware- und Softwarearchitektur ab. Aufbauend auf den grundlegenden Konzepten und Begriffen aus Kapitel 2.3.1 konzentriert sich dieser Abschnitt auf die praktische Implementierung. Im Mittelpunkt stehen die Auswahl geeigneter Hardwarearchitekturen sowie die Anwendung passender Programmiermodelle, um die theoretischen Grundlagen effizient in die Praxis zu übertragen. Die Flynn´sche Taxonomie ist eine grundlegende Methode zur Klassifikation von Computersystemen. Sie kategorisiert Computersysteme basierend auf der Anzahl von gleichzeitig ausgeführten Befehls- und Datenströmen in vier Hauptkategorien (Hennessy und Patterson, 2011, S. 11):

- **SISD (Single Instruction, Single Data):** Traditionelle sequentielle Computer, die eine einzige Verarbeitungseinheit auf einen einzigen Datenstrom anwenden.
- **SIMD (Single Instruction, Multiple Data):** Systeme, die denselben Befehl gleichzeitig auf mehrere Datenströme anwenden. Dies ist typisch für Vektorprozessoren und Graphics Processing Units (GPUs), die für datenparallele Aufgaben wie Grafikverarbeitung oder wissenschaftliche Berechnungen verwendet werden.
- **MISD (Multiple Instruction, Single Data):** Eine seltene Kategorie, bei der mehrere Befehle auf denselben Datenstrom angewendet werden.
- **MIMD (Multiple Instruction, Multiple Data):** Systeme, in denen mehrere Befehlsströme auf mehrere Datenströme angewendet werden. Dies ist typisch für Multiprozessorsysteme und verteil-

te Systeme, bei denen verschiedene Prozessoren unabhängig voneinander unterschiedliche Programme auf unterschiedlichen Daten ausführen.

Viele parallele Prozesse verwenden eine hybride Form dieser Klassifikation. Neben der Klassifikation des Computersystems spielt die Wahl der Speicherarchitektur eine zentrale Rolle, da sie die Art der Datenkommunikation und Synchronisation zwischen den parallel arbeitenden Einheiten bestimmt. Es werden dabei zwei Grundtypen unterschieden (B. Wilkinson und Allen, 2005, S. 13f.):

- **Shared-Memory-Architektur:** In der Shared-Memory-Architektur teilen sich mehrere Prozessoren einen gemeinsamen Speicher. Die Kommunikation findet direkt über den Speicher statt, ohne explizite Nachrichten zwischen Prozessoren senden zu müssen. Die Programmierung wird dadurch vereinfacht, aber Speicherzugriffskonflikte können die Skalierbarkeit einschränken. Deshalb sind Synchronisationsmechanismen, wie Locks, Semaphoren und Barrieren, notwendig, um Datenkonflikte zu vermeiden.

- **Distributed-memory Architektur:** In einer Distributed-memory Architektur hat dagegen jeder Prozessor einen eigenen Speicher. Die Prozessoren kommunizieren über ein Netzwerk durch den Austausch von Nachrichten, dem sogenannten Message Passing. Diese Architektur bietet bessere Skalierbarkeit und weniger Speicherzugriffskonflikte, ist jedoch komplexer zu programmieren und es kommt häufiger zu dem in Kapitel 2.3.1 genannten Kommunikations-Overhead.

Die von B. Wilkinson und Allen, 2005 beschriebenen Architekturen bilden die Basis für für die Implementierung verschiedener Programmiermodelle und -paradigmen, die unabhängig von den spezifischen Hardwareanforderungen allgemeine Prinzipien paralleler und verteilter Verarbeitung ermöglichen. Für Shared-Memory-Architekturen wird häufig OpenMP genutzt, das mithilfe von Compiler-Direktiven die parallele Ausführung von Schleifen und Codeblöcken steuert. Es ermöglicht schnelle Kommunikation zwischen Prozessoren, erfordert jedoch Synchronisationsmechanismen wie Locks oder Barrieren. Bei Distributed-Memory-Architekturen sind Message Passing Interface (MPI) und das Bulk Synchronous Parallel (BSP)-Modell verbreitete Ansätze. MPI ermöglicht die Kommunikation zwischen Prozessoren über Nachrichten und ist besonders für Anwendungen auf Clustern und im HPC geeignet. Für verteilte Big Data-Anwendungen in Cloud-Umgebungen ist MapReduce ein zentrales Programmiermodell (Parhami, 2018, S. 3). Es basiert auf einer Distributed-Memory-Architektur und teilt Daten in Partitionen auf, die parallel verarbeitet werden. In der Map-Phase entstehen Schlüssel-Wert-Paare, die in der Reduce-Phase aggregiert werden, um das Endergebnis zu erzeugen (Bengel et al., 2015, S. 99). MapReduce ist dank der Skalierbarkeit der Cloud besonders gut für Big-Data-Anwendungen geeignet, wie die Verarbeitung großer Geodatenmengen (Ji et al., 2012, S. 17).

2.4 Parallele und verteilte Verarbeitung mit Dask

Dask ist ein Framework für parallele und verteilte Datenverarbeitung, das speziell für große Datenmengen entwickelt wurde. Es bietet eine nahtlose Integration in das bestehende Python-Ökosystem und ermöglicht es, große und komplexe Datenverarbeitungsaufgaben sowohl auf Einzelrechnern als auch in verteilten Clustern effizient zu bewältigen (Dask core developers, 2024, o.S.). Damit konkurriert Dask direkt mit Apache Spark, einem weiteren führenden Framework für verteilte Datenverarbeitung. Während beide Systeme darauf abzielen, die Last großer Datenmengen auf mehrere Knoten zu verteilen, bietet Dask eine tiefere Integration in das Python-Ökosystem. Die Bibliothek zeichnet sich durch eine hohe Flexibilität aus und bietet eine einfache API, die sich mit vielen bekannten Python-Verarbeitungsbibliotheken kombinieren lässt. Dazu zählen beispielsweise Bibliotheken für Core (SciPy), Datenvorbereitung (NumPy, Pandas), Datenvisualisierung (Matlibplot), maschinelles Lernen (Scikit-Learn) und Deep Learning (PyTorch, TensorFlow). Dask ermöglicht die Verarbeitung großer Datensätze sowohl auf Einzelrechnern als auch in Clustern und wird in Bereichen wie Datenanalyse, maschinellen Lernen und wissenschaftlichen Berechnungen eingesetzt (Böhm und Beránek, 2020, S. 1). Es bietet im Vergleich zu Apache Spark eine tiefere Python-Integration und Flexibilität bei der Nutzung bekannter Bibliotheken (Peters, 2023, S.5).

Bedeutende Organisationen wie NASA, NVIDIA und Microsoft setzen Dask für die Analyse großer Datenmengen, Satellitenbildverarbeitung und skalierbare ETL-Pipelines ein (Dask core developers, 2024, o.S.). Die NASA nutzt es in Kombination mit Xarray zur effizienten Verarbeitung raumzeitlicher Rasterdaten. Dask eignet sich hier besonders, da es n-dimensionale Array-Operationen unterstützt und Daten handhaben kann, die nicht in den Arbeitsspeicher eines einzelnen Rechners passen. Durch die parallele Verarbeitung und Verteilung reduziert es die Rechenzeit erheblich und erweist sich daher als ideales Werkzeug für diese Untersuchung.

2.4.1 Architektur und Komponenten

Die Architektur von Dask ist so konzipiert, dass sie sowohl eine parallele als auch verteilte Verarbeitung ermöglicht. Durch Multithreading können mehrere Threads verwendet werden, um Aufgaben gleichzeitig auszuführen, was besonders bei I/O-gebundenen Aufgaben nützlich ist. Bei CPU-gebundenen Aufgaben kann Dask mittels Multiprocessing mehrere Prozesse verwenden, um die Berechnungen parallel auszuführen und so die Leistung zu steigern. Für noch größere Datenmengen und komplexere Berechnungen kann Dask in einem verteilten Cluster eingesetzt werden. Dies ermöglicht die Skalierung von Berechnungen über mehrere Maschinen hinweg. Dadurch können Datensätze verarbeitet werden, die größer sind als der verfügbare Speicher und komplexe Berechnungsaufgaben viel schneller ausgeführt werden als mit herkömmlichen Methoden.

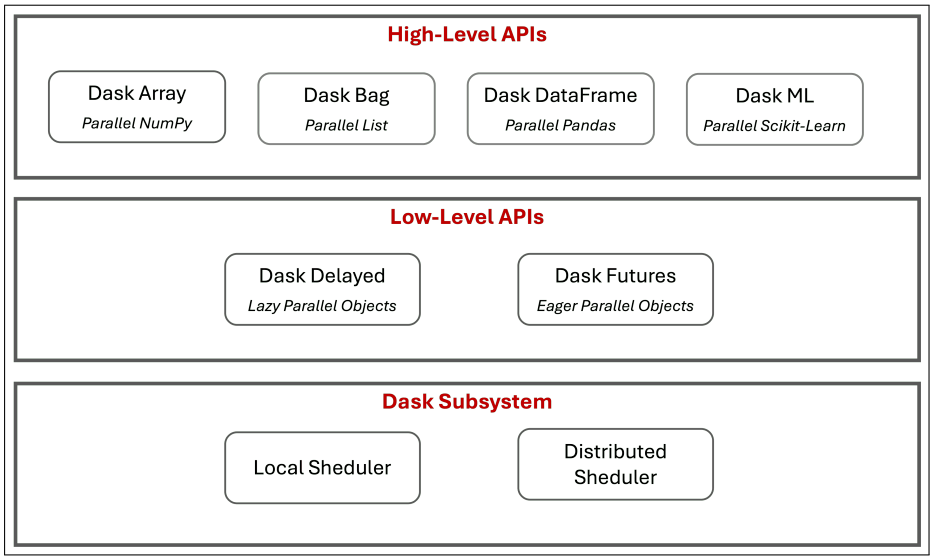

Abbildung 4: Komponenten von Dask (Eigene Darstellung nach Petrelli, 2023, S. 163)

Wie in Abbildung 4 dargestellt, besteht Dask grundsätzlich aus drei Hauptschichten: (1) Scheduler, (2) Low-Level-API und (3) High-Level-API. Im Kern steht der Task-Scheduler, der die Ausführung von Berechnungen über CPU-Kerne und Maschinen hinweg überwacht. Diese Berechnungen werden im Python-Code entweder als Dask Delayed-Objekte, Dask Futures-Objekte oder als konkrete Werte dargestellt. Delayed-Objekte werden „lazy“ ausgewertet. Das bedeutet, dass die Auswertung erst stattfindet, wenn die Werte benötigt werden. Future-Objekte werden dagegen „eagerly“ ausgewertet. Die Verarbeitung erfolgt in Echtzeit, unabhängig davon, ob der Wert sofort benötigt wird oder nicht (Daniel, 2019, S. 7). Dasks High-Level-APIs bieten eine Abstraktionsschicht über Delayed- und Futures-Objekte. Operationen auf diesen High-Level-Objekten führen zu vielen parallelen Low-Level-Operationen, die von den Task-Schedulern verwaltet werden. Um die tatsächliche Berechnung von konkreten Werten auszuführen, werden drei Methoden unterschieden (Anaconda und Contributors, 2016, o.S.):

- **Compute:** Dask.compute() berechnet alle verzögerten Operationen sofort und lädt die Ergebnisse in den lokalen Speicher.

- **Persist:** Dask.persist() hält die Berechnungen im Speicher, um sie für weitere Berechnungen zwischengespeichert bereitzuhalten, ohne sie direkt an den lokalen Speicher zu übergeben.

Die High-Level-API stellen die grundlegenden Datenstrukturen dar. Diese unterstützen eine große Teilmenge der Funktionen, die auch von ihren äquivalenten Bibliotheken bereitgestellt werden. Darüber hinaus sind sie dahingehend optimiert, diese Operationen auf parallele Weise auszuführen. Folgende Datenstrukturen werden dabei unterschieden:

- **Dask-Array:** Dask-Arrays kombinieren viele NumPy-Arrays, die in Blöcken innerhalb eines Rasters angeordnet sind. Sie entsprechen daher der parallel-freundlichen Version von NumPy-Arrays, da die Blöcke unabhängig voneinander verarbeitet werden können. Dies ermöglicht die parallele Ausführung numerischer Berechnungen über mehrere Prozessoren hinweg (Petrelli, 2023, S. 162).
- **Dask-Bag:** Dask-Bag ist für die Verarbeitung von unstrukturierten oder semi-strukturierten Daten optimiert und ähnelt Listen in Python. Es ist besonders nützlich für Aufgaben wie Textverarbeitung (Anaconda, Inc. and Contributors, 2018b, o.S.).
- **Dask-Dataframe:** Ein Dask-DataFrame ist das Gegenstück zu einem Pandas Dataframe. Es besteht aus vielen kleineren Dataframes, die entlang eines Index aufgeteilt sind. Dask-Dataframes sind besonders nützlich für Datenanalysen und -manipulationen auf großen Datensätzen (Petrelli, 2023, S. 164).
- **Dask-ML:** Dask-ML bietet skalierbares maschinelles Lernen in Python mit Dask zusammen mit beliebten Machine-Learning-Bibliotheken wie Scikit-Learn, XGBoost und anderen. Dabei löst Dask die Skalierung der Modellgröße und der Datengröße mit Hilfe von High-Level-APIs und Dask-Clustern, um Aufgaben wie Modelltraining, Vorhersage oder Bewertungsschritte möglichst performant durchführen zu können. Durch die Integration mit beliebten Bibliotheken wie Scikit-Learn können komplexe ML-Pipelines erstellt werden, die parallel über mehrere CPUs oder GPUs ausgeführt werden (Dask Developers, 2017, o.S.).

Die Objekte aus den High-Level- und Low-Level-API werden auch *Collections* genannt. Sobald Operationen auf diesen Collections ausgeführt werden, wird ein Task-Graph erstellt, der die Berechnung darstellt (siehe Abbildung 5). Ein Task-Graph besteht aus einer Reihe von Aufgaben mit Abhängigkeiten zwischen ihnen, wobei jede Aufgabe eine kleine, deterministische Funktion ist, die Eingaben entgegennimmt und Ausgaben erzeugt (Peters, 2023, S. 12). Ein Beispiel ist die Bestimmung des höchsten Pixelwertes in einem großen Rasterbild, das als Digitales Höhenmodell vorliegt. Dask nimmt das Höhenraster und teilt es in kleinere Abschnitte auf. Daraus wird ein Task-Graph erstellt, der festlegt, wie der höchste Pixelwert in jedem dieser Abschnitte ermittelt wird und wie diese Teilresultate später kombiniert werden, um den höchsten Wert im gesamten Bild zu finden. Der Task-Graph zeigt den gesamten Ablauf der Berechnung, führt ihn aber nicht sofort aus. Die Ausführung des Aufgabendiagramms wird von einem Dask-Scheduler übernommen. Der Scheduler nimmt das Aufgabendiagramm und führt die Aufgaben in einer Reihenfolge aus, die ihre Abhängigkeiten berücksichtigt. Bei der Höhenwertbestimmung würden beispielsweise die Aufgaben zur Bestimmung der maximalen Höhe der einzelnen Teile des Arrays vor der Aufgabe zur Kombination dieser Höhenwerte ausgeführt werden. Die Scheduler von Dask sind für die parallele Ausführung von Aufgabendiagrammen konzipiert. Sie können Aufgaben auf mehreren Threads oder Prozessen auf einer einzelnen Maschine oder auf mehreren Maschinen in einem Cluster ausführen. Die Scheduler sorgen für den Lastausgleich zwischen verschiedenen Workern und die Wiederherstellung bei Ausfällen von Workern.

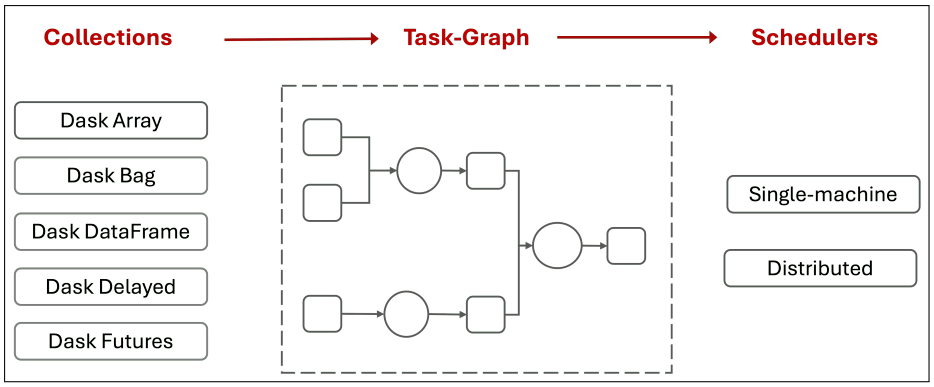

Abbildung 5: Dask-Workflow (Eigene Darstellung nach Anaconda, Inc. and Contributors, 2018a, o.S.)

Dask unterstützt mehrere Backends für die Berechnung, wobei in dieser Arbeit ausschließlich Dask.distributed verwendet wird. Die Architektur besteht aus drei Hauptkomponenten, deren Zusammenhang in Abbildung 6 schematisch dargestellt ist.

- **Client:** Der Client ist die benutzerseitige API, die zur Ausführung verteilter Berechnungen verwendet wird. Der Client-Code stellt eine Verbindung zu einem Dask-Cluster her, sendet Aufgabengraphen an den Server und sammelt die Ergebnisse.
- **Server/Scheduler:** Der Server ist die zentrale Komponente des Dask-Clusters, der die Worker koordiniert und die Anfragen der Clients bearbeitet. Er enthält einen Scheduler, der den Workern nach einer speziellen Heuristik Aufgaben zuweist, um die Last des Clusters auszugleichen. Wenn ein Ungleichgewicht auftritt (einige Worker sind unter-/überlastet), versucht der Scheduler, Aufgaben von überlasteten Knoten zu stehlen und sie auf unterlastete Knoten zu verteilen. Der Scheduler weist den Aufgaben auch Prioritäten zu, die von den Workern verwendet werden, um zu entscheiden, welche Aufgaben zuerst berechnet werden sollen.
- **Worker:** Ein Worker ist ein Prozess, der die vom Client übermittelten Aufgaben ausführt. Der Server sendet Aufgaben (bestehend aus serialisierten Python-Funktionen und Argumenten) an einzelne Worker, die sie ausführen. Die Worker kommunizieren untereinander, um Ausgaben für Aufgaben auszutauschen, die lokal nicht verfügbar sind. Jeder Worker ist mit einer bestimmten Anzahl von CPU-Kernen konfiguriert, die er verwenden darf. Die Worker verarbeiten ihre Aufgaben parallel, führen aber nie mehr als eine Aufgabe pro verfügbarem Kern gleichzeitig aus (Böhm und Beránek, 2020, S. 2).

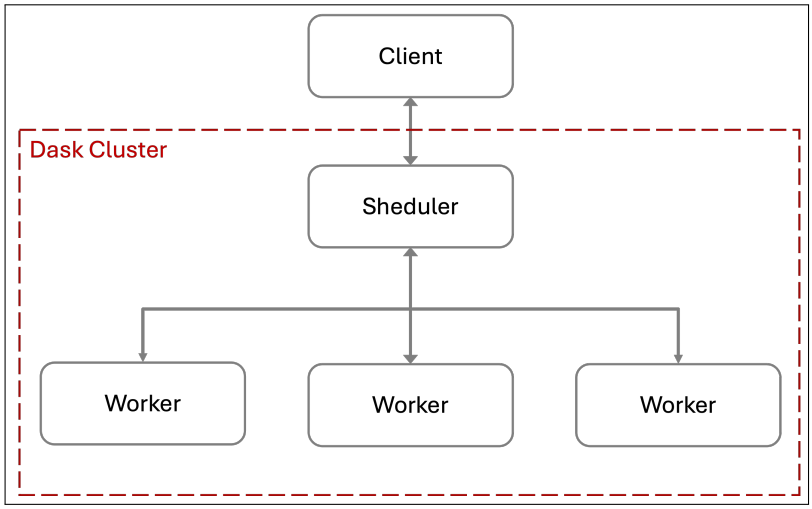

Abbildung 6: Aufbau des Distributed-Scheduler in Dask (Eigene Darstellung nach Dask Developers, 2018, o.S.)

2.4.2 Deployment

Bei der Verwendung von Dask wird, wie in Abbildung 4 dargestellt, zwischen lokalem Umfeld und einem verteilten System unterschieden. Auf einer lokalen Maschine kann Dask ohne spezielle Konfiguration betrieben werden, wobei standardmäßig Threads verwendet werden, um parallele Berechnungen durchzuführen (Anaconda, Inc. and Contributors, 2018d, o.S.). Alternativ kann ein Multi-Prozess-Cluster eingerichtet werden, das die parallele Verarbeitung über mehrere Prozesse hinweg ermöglicht. Dabei wird ein Scheduler und mehrere Worker-Prozesse auf der lokalen Maschine gestartet. Dies ermöglicht es, parallele Berechnungen im kleinen Rahmen zu testen.

Für größere und rechenintensivere Aufgaben ist der Einsatz eines verteilten Dask-Clusters erforderlich. Dieser kann in einer Vielzahl von verteilten Umgebungen bereitgestellt werden, darunter lokale HPC-Cluster, Cloud-Plattformen wie AWS oder Azure sowie Container-Orchestrationsplattformen wie Kubernetes (Anaconda, Inc. and Contributors, 2018d, o.S.). Insbesondere Cloud-Plattformen bieten die Möglichkeit, Dask-Cluster schnell und effizient zu skalieren. Die Verwaltung der Cloud-APIs und Softwareumgebungen kann jedoch eine Herausforderung darstellen, weshalb kommerzielle Lösungen wie Coiled oder Open-Source-Optionen wie der Dask Cloud Provider hilfreich sein können.

Deployment auf Kubernetes
Kubernetes ist eine beliebte Plattform für die Bereitstellung verteilter Anwendungen und eignet sich gut für die Einrichtung von Dask-Clustern. Die Einrichtung kann über den Dask-Kubernetes-Operator oder ein Dask Gateway vollzogen werden (Anaconda, Inc. and Contributors, 2018d, o.S.). Der Kubernetes-Operator ist eine Kubernetes-native Lösung, die es ermöglicht, Dask-Cluster als Custom Resource Definitions (CRDs) in bestehende Umgebungen zu integrieren. Diese Option richtet sich an erfahrene Nutzer, die volle Kontrolle über ihre Cluster-Deployments benötigen. Das Dask Gateway hingegen ist besonders für Umgebungen mit mehreren Nutzern geeignet, die Dask-Cluster nutzen, ohne direkten Zugriff auf die zugrunde liegende Infrastruktur zu haben.

Dask Gateway
Dask Gateway dient als Vermittler zwischen den Benutzern und der Cluster-Infrastruktur. Es ermöglicht mehreren Benutzern, in einer gemeinsamen Umgebung eigene Dask-Cluster zu erstellen und darauf zuzugreifen, ohne dass sie administrative Rechte oder Kenntnisse über die zugrunde liegende Infrastruktur benötigen. Die Installation von Dask Gateway kann beispielsweise wie in Kapitel 2.2 beschrieben über ein Helm-Chart vorgenommen werden (Jim Crist-Harif, 2021, o.S.). In Abbildung 7 werden die Hauptkomponenten des Dask Gateways dargestellt.

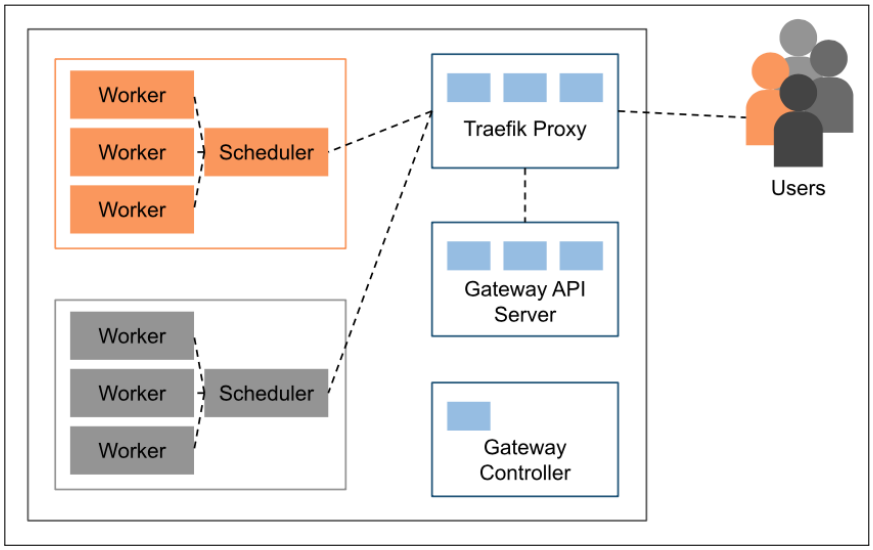

Abbildung 7: Deployment eines Dask-Clusters in Kubernetes (Jim Crist-Harif, 2021, o.S.)

Ein zentrales Element des Dask Gateway ist der ***Gateway API Server***, der als Schnittstelle zwischen den Benutzern und der Cluster-Verwaltung dient. Benutzer können über den API Server Anfragen stellen, um eigene Dask-Cluster zu starten. Nachdem ein Benutzer über den API Server ein neues Cluster angefordert hat, übernimmt der ***Gateway Controller*** die Verwaltung dieser Anfrage. Dieser hat die erforderlichen Rechte, um im Kubernetes-Cluster neue Pods zu erstellen oder zu entfernen und die Ressourcen zu verwalten und zu skalieren. Um sicherzustellen, dass die Kommunikation zwischen den Benutzern und ihren individuellen Dask-Clustern reibungslos funktioniert, wird ein ***Traefik Proxy*** eingesetzt. Dieser fungiert als Lastverteiler und leitet die Anfragen der Benutzer an das jeweilige Cluster weiter. So können mehrere Benutzer gleichzeitig und unabhängig voneinander Berechnungen durchführen, ohne dass es zu Konflikten kommt. Ein Dask-Cluster besteht typischerweise aus einem ***Scheduler*** und mehreren ***Workern***. Der Scheduler ist dabei für die Verteilung der Berechnungsaufgaben an die Worker des Clusters zuständig, die dann die eigentlichen Rechenoperationen ausführen. In Kubernetes-Umgebungen werden diese als Pods aufgeführt. Beispielsweise können dort Pods mit den Namen „dask-scheduler-..“ und „dask-worker-...“ identifiziert werden, die zu einem Dask-Cluster gehören. Ein solches Cluster kann dynamisch skaliert werden, indem die Anzahl der Worker während der Laufzeit erhöht oder verringert wird. Kubernetes stellt die erforderlichen Ressourcen entsprechend durch das Starten neuer Worker-Pods oder Beenden bestehender Pods bereit.

2.4.3 Diagnostik und Optimierung

Die effiziente Nutzung und Optimierung der Dask-Infrastruktur ist entscheidend, um die Leistung bei rechenintensiven Anwendungen zu maximieren und Engpässe zu identifizieren. Hierzu stellt Dask eine Reihe von diagnostischen und Optimierungswerkzeugen bereit, die speziell auf die verteilte Architektur des Frameworks ausgelegt sind. Zu den wichtigsten gehören das Dask-Dashboard, Task-Graph-Visualisierungen und der Performance-Report.

Das Dask-Dashboard ist das zentrale Werkzeug zur Überwachung von Systemressourcen und zur Diagnose verteilter Berechnungen, auf die über einen Webbrowser zugegriffen werden kann. Über diese Schnittstelle können Echtzeitinformationen über den CPU- und Speicherverbrauch, Netzwerkverkehr und zur Auslastung einzelner Worker überwacht werden. Zu den wichtigsten Diagnoseelementen des Dashboards gehören (Peters, 2023, S. 47f.):

- **Task Stream-Visualisierung:** Dieses Tool zeigt den laufenden Fortschritt der Berechnungen an, um die Reihenfolge und den Status von Aufgaben zu überwachen und Engpässe frühzeitig zu erkennen.
- **Progress Bar:** Der Fortschrittsbalken zeigt auf einfache Weise den Anteil der erledigten Aufgaben zum Gesamtfortschritt an.
- **Memory Usage und CPU Usage:** Diese Funktionen bieten detaillierte Einblicke in den Speicher- und CPU-Verbrauch der einzelnen Worker und ermöglichen es, ressourcenintensive Aufgaben zu identifizieren und das Speichermanagement gezielt zu optimieren.

Die Farbcodierungen im Dahsboard geben dabei beispielsweise Aufschluss, wann die Worker die Schwellenwerte für die Speichernutzung überschreiten oder welche Aufgabenarten besonders viel Zeit beanspruchen. Ein weiteres bedeutendes Tool in der Dask-Diagnostik ist Graphviz, das eine grafische Darstellung der Task-Graphen von Dask ermöglicht. Diese Visualisierung gibt Einblick in die Struktur Abhängigkeiten und Reihenfolge der Berechnungen und ermöglicht es, Bereiche mit redundanten oder überflüssigen Aufgaben zu identifizieren. Mit dem Performance-Report können HTML-Übersichten generiert werden, die den Aufgabenstrom, die Aufgabendauer, die Speichernutzung und die CPU-Auslastung der Worker im Zeitverlauf darstellt. Mit einem MemorySampler können Plots zur Speichernutzung des Clusters während der Aufgabenausführung erstellt werden. Zudem bietet Dask Best Practices für das Speichermanagement und die Einstellung von Chunkgrößen, um die Auslastung zu optimieren (Anaconda, Inc. and Contributors, 2018c, o.S.).

3 Raumzeitliche Rasterdaten

Raumzeitliche Rasterdaten repräsentieren raumzeitliche Phänomene, Ereignisse oder Prozesse, die sich über eine bestimmte räumliche und zeitliche Ausdehnung und unterschiedliche Skalen erstrecken (Yuan und Mcintosh, 2002, S. 14). Diese Phänomene können Ereignisse (z.B. Waldbrände), Zustände (z.B. Temperaturverteilungen) oder Positionen von Objekten über die Zeit abbilden. Viele Datenquellen liefern ihre Daten per se im Rasterdatenmodell. Zu den gängigen Datenquellen zählen Satelliten- und Luftbilder, gescannte Karten oder digitale Bilder, die an festen räumlichen Positionen und zu verschiedenen Zeitpunkten gesammelt werden. Wenn Rasterdaten mit einem Zeitstempel erfasst werden, wird von raumzeitlichen Rasterdaten gesprochen (Alam et al., 2022, S. 8). Solche Daten umfassen typischerweise:

- Räumliche Dimension (z.B. Koordinaten)
- Zeitlichen Informationen (z. B. Zeitstempel oder Zeitspannen)
- Nicht-räumliche Informationen (z. B. Temperatur, Höhe oder spektrale Informationen)

Je nach Art der abgebildeten Daten können Rasterdaten diskrete Informationen wie die Bodenart oder kontinuierliche Variablen wie Temperatur oder Höhenverteilungen darstellen (Alam et al., 2022, S. 7). Die Darstellung erfolgt in Form einer Matrix von Zellen (Pixel), wobei jede Zelle einen Wert besitzt, der die nicht-räumliche Information enthält (Bill, 2016, S. 31). Diese Werte können gemessene Größen wie die Höhe oder spektrale Eigenschaften umfassen. Durch die Angabe von Zeile und Spalte sind die Informationen über die Geoobjekte identifizierbar. Über die Lage des linken oberen Pixels, der Zellengröße in x- und y-Richtung sowie der Anzahl der Pixel definiert sich die Lage, Größe und Ausdehnung des Datensatzes. Die Anzahl der Pixel pro Flächeneinheit bestimmt die räumliche Auflösung, während die Häufigkeit der Datenerfassung die zeitliche Auflösung bestimmt. Eine höhere räumliche Auflösung bedeutet kleinere Zellen und damit eine genauere Darstellung, während eine höhere zeitliche Auflösung, wie stündliche statt tägliche Messungen, genauere Analysen zeitlicher Veränderungen ermöglicht. Typischerweise werden solche Daten als mehrdimensionale Arrays gespeichert.

Zu den Vorteilen von Rasterdaten zählen ihre einfache Datenstruktur und ihre Eignung zur Darstellung kontinuierlicher Phänomene. Die zeitliche Dimension spielt eine entscheidende Rolle, um Veränderungen im Zeitverlauf zu erfassen und zu analysieren. Die Kombination von Raum und Zeit in solchen Daten erlaubt es, dynamische Prozesse besser zu verstehen und vorherzusagen. Deshalb werden raumzeitliche Rasterdaten in vielen Bereichen wie Fernerkundung, Umweltwissenschaften, Stadtplanung und Klimaforschung eingesetzt. Allerdings steigen die Datenmengen mit höherer Auflösung exponentiell an, was eine leistungsfähige Infrastruktur zur Speicherung und Verarbeitung erfordert. Zudem erschwert die Zellenstruktur die präzise Abgrenzung von Objekten, und geometrische Operationen sind oft nur eingeschränkt möglich (Bill, 2016, S. 31). Hinzu kommt, dass raumzeitliche Rasterdaten oft Unsicherheiten in der zeitlichen Erfassung aufweisen, beispielsweise durch Wolkenbedeckung bei Satellitenaufnahmen, die zu Lücken in den Zeitreihen führen können. Solche Daten müssen vor der Analyse häufig bereinigt und interpoliert werden, um konsistente zeitliche Verläufe zu gewährleisten.

3.1 Repräsentation raumzeitlicher Rasterdaten

Die Integration der Zeitdimension in raumzeitliche Rasterdaten ist von zentraler Bedeutung, um Veränderungen und Dynamiken von Phänomenen über Raum und Zeit hinweg zu erfassen und zu analysieren. Die übergeordneten Ziele sind, Trends und Vorhersagen abzuleiten, Ursachen zu verstehen und Entscheidungsfindungen zu unterstützen. Um diese greifbar und verständlich zu machen, sind raumzeitliche Modelle notwendig, die Veränderungen nicht nur im Raum, sondern auch in ihrem zeitlichen Verlauf abbilden können. Auf diese Weise können Informationen und Erkenntnisse aus den Datenmengen durch Analysen und Visualisierungen abgeleitet werden (siehe Abbildung 8).

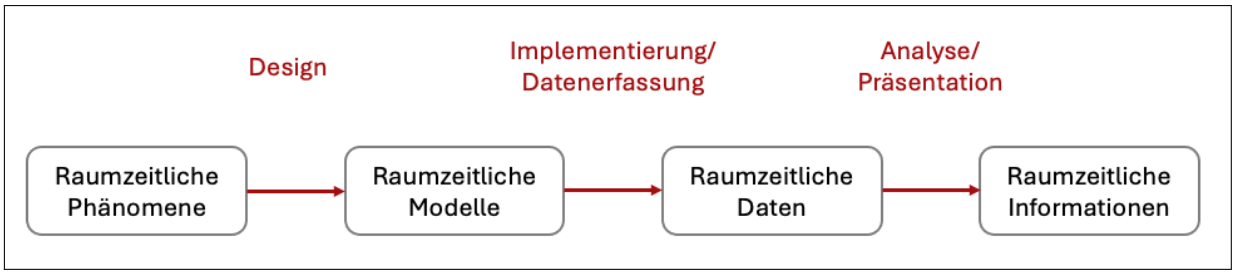

Abbildung 8: Repräsentation raumzeitlicher Phänomene (Eigene Darstellung)

Im Laufe der Zeit haben sich zahlreiche Modelle entwickelt, die unterschiedliche Ansätze zur Integration von Raum und Zeit bieten. Die Wahl des passenden Modells hängt von der Fragestellung und dem Ziel der Analyse ab. Einige Modelle fokussieren sich darauf, Veränderungen zu identifizieren, andere wollen Ursachen für diese Veränderungen finden oder die Prozesse, die den Veränderungen zugrunde liegen, abbilden. Zur Klassifikation schlagen Carré und Hamdani, 2021 eine pyramidale Hierarchie vor, die die Komplexität raumzeitlicher Modelle in sechs aufeinander aufbauenden Stufen abbildet – von einfachen Momentaufnahmen (Snapshots) über objekt- und ereignisbasierten Modellen bis hin zu Modellen, die kausale Zusammenhänge erfassen (Carré und Hamdani, 2021, S. 5). Ein Beispiel ist die Beobachtung von Veränderungen in einem Waldgebiet. Ein einfaches Snapshot-Modell stellt dies als Serie von Rasterkarten dar, die die Waldbedeckung zu verschiedenen Zeitpunkten zeigen. Veränderungen zwischen zwei Zeitpunkten werden meist durch Klassifikationsalgorithmen identifiziert und können etwa auf eine Verkleinerung der Waldfläche hinweisen. Solche Modelle bieten jedoch oft nur eine vereinfachte Darstellung von Änderungen und deren Ursachen. Komplexere Ansätze hingegen ermöglichen die Nachverfolgung spezifischer Merkmale wie Größe, Dichte und Position von Baumgruppen oder die Analyse konkreter Ereignisse, etwa Schädlingsbefall oder menschliche Eingriffe, um Veränderungen umfassender zu verstehen.

Im Bereich der Geoinformation wird jedoch verstärkt die räumliche Komponente betrachtet, wobei die zeitliche Dimension oft auf die Frage reduziert wird, ob es Änderungen gibt. Für die Verarbeitung und Analyse raumzeitlicher Rasterdaten haben sich in der Praxis insbesondere vier Datenmodelle etabliert: Szenen, Analysis Ready Data (ARD), Data Cubes und Composite-Layer. Diese Modelle basieren auf Snapshots und Data Cubes und sind speziell darauf ausgelegt, raumzeitliche Rasterdaten effizient bereitzustellen und zu analysieren. Diese Modelle erlauben deskriptive Analysen zu grundlegenden Veränderungen über die Zeit, sind jedoch in ihrer Fähigkeit begrenzt, tiefere kausale Beziehungen und dynamische Wechselwirkungen abzubilden (Kavouras, 2001, S. 8f.). Der hierarchische Modellansatz von Carré und Hamdani, 2021 ordnet solche Snapshots und Data Cubes deshalb in die unteren Ebenen ein.

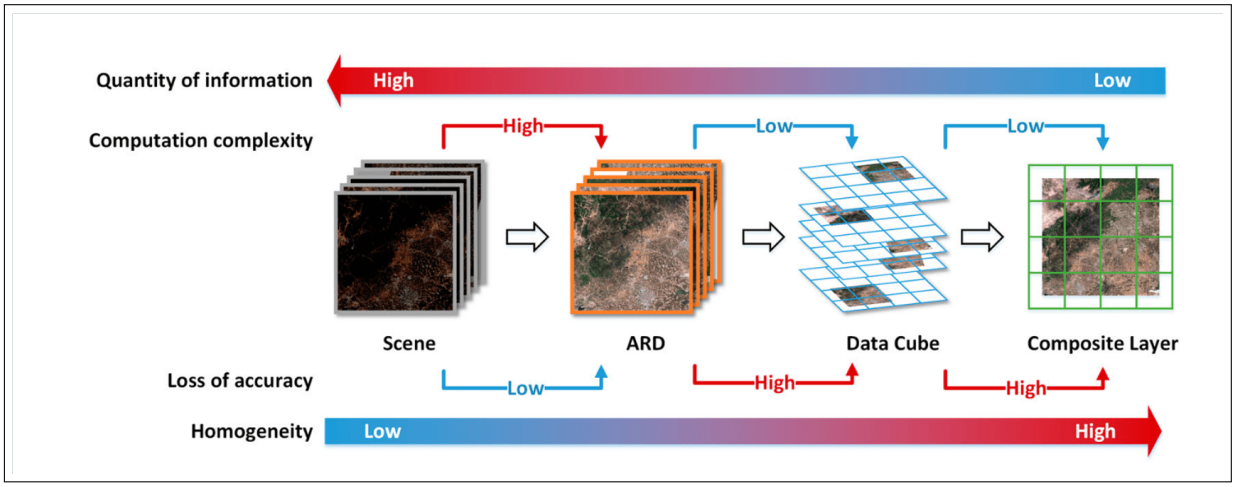

Abbildung 9: Modellierung von raumzeitlichen Rasterdaten (Xu, Du, Fan et al., 2022, S. 1427)

In Abbildung 9 werden die Datenmodelle Szenen, ARD, Data Cubes und Composite Layer in Bezug auf Informationsmenge, Homogenität, Rechenkomplexität und Genauigkeit dargestellt. Je weiter sich von losen Szenen zu einem Composite-Layer bewegt wird, desto höher werden Homogenität und Datenzugänglichkeit, während die Datenmenge und Rechenkomplexität tendenziell abnimmt. Rasterdaten, wie Satelliten- und Luftbilder, liegen zunächst häufig als lose ***Szenen*** vor, die den Zustand eines räumlichen Gebietes zu einem bestimmten Zeitpunkt darstellen. Für kleinräumige Analysen reichen diese Szenen häufig aus. Die Notwendigkeit anderer Datenmodelle ergibt sich vorallem aus den Herausforderungen mit raumzeitlichen Rasterdaten. Durch die explosionsartigen Zunahme raumzeitlicher Rasterdaten kommt eine hohe Informationsdichte zustande, die allerdings mit einem hohen Speicherbedarf dieser Datenmengen verbunden ist, der oftmals über die Verfügbarkeit lokaler Rechenressourcen hinausgeht. Zudem liegen die Daten häufig in verschiedenen räumlichen, zeitlichen und spektralen Auflösungen vor. Auch die Qualität der Erdbeobachtungsdaten ist aufgrund unterschiedlicher Wolken- und Lichteinflüsse zu verschiedenen Befliegungszeitpunkten inhomogen. Dies erschwert den Zugriff und die Nutzung raumzeitlicher Rasterdaten bei der Informationsextraktion. Die unvorverarbeiteten Szenen erfordern daher eine hohe Rechenleistung, da sie ohne Vorbereitung schwer zu integrieren sind.

Die Vorbereitung von Rasterdaten ist deshalb insbesondere bei großflächigen Analysen ein wichtiger Schritt. Ein Aspekt ist dabei die Homogenität, d.h. die Einheitlichkeit von Eigenschaften wie Auflösung, Koordinatensystem oder spektralen Informationen. Deshalb werden die Szenen zu ***ARD*** verarbeitet, was bedeutet, dass die Daten vorbereitet und standardisiert werden (siehe Abbildung 9). Zu der Vorprozessierung zählen Arbeitsschritte wie Resampling, radiometrische und geometrische Korrekturen, Filterungen, Koordinatentransformationen, Tiling, um parallele Verarbeitung zu erlauben und die Entfernung von Überlappung (Kopp et al., 2019, S. 4ff.). Diese Vorverarbeitung erhöht die Zugänglichkeit, indem sie die Daten in ein einheitliches Format bringt. Allerdings ist die Umwandlung von Szenen zu ARD ein aufwendiger Prozess.

Für raumzeitliche Analysen, die über längere Zeiträume aggregieren, werden ARD oft in ***Data Cubes*** überführt. Ein Data Cube kombiniert ARD räumlich und zeitlich in einem konsistenten Data Cube, der eine organisierte, mehrdimensionale Darstellung der Daten bietet. Die Erstellung von ARD ist dabei ein fundamentaler Schritt zur Bildung von Data Cubes, wie die Open Data Cube Initiative herausstellt, um die Daten für Nicht-Experten verfügbar zu machen und die Nutzung der Daten zu erleichtern (Kopp et al., 2019, S. 3). Die Umwandlung von ARD zu Data Cubes ist weniger aufwendig, da sie vor allem eine Reorganisation der Daten über die Zeit darstellt. Xu, Du, Fan et al., 2022 argumentiert, dass die Erstellung von Data Cubes zu einem hohen Genauigkeitsverlust führen kann. Dies ist jedoch variabel, da im einfachsten Fall lediglich eine Reorganisation der Daten über die Zeit erfolgt, wodurch kein Verlust entsteht. Häufig wird jedoch bei der Erstellung eines Data Cubes der Fokus auf benutzerdefinierte Anwendungen gelegt. Dabei kommen häufig Reprojektionen, Resampling, Filterung oder Aggregationen zum Einsatz. Beispielsweise kann ein Informationsverlust bei der Aggregation von monatlichen zu jährlichen Mittelwerten auftreten, da feine zeitliche Details verloren gehen. Auch die Harmonisierung unterschiedlicher räumlicher Auflösungen bei der Integration verschiedener Datensätze kann zu einer Informationsreduktion führen. Je nach Spezialisierungsgrad kann der Übergang von ARD zu Data Cube dementsprechend aufwendiger werden.

Composite-Layer sind eine generalisierte Darstellung der Daten, bei der die Informationsmenge niedrig ist, die Homogenität jedoch hoch. Er kann verwendet werden, wenn es weniger um die zeitliche Dynamik, sondern eher um eine aggregierte Darstellung und Visualisierung geht, wie z. B. Jahresmittelwerte. Allerdings geht auf diesem Level oft auch Genauigkeit verloren, da die Daten vereinfacht, selektiert und zusammengefasst werden (Xu, Du, Fan et al., 2022, S. 1428). Insgesamt enthalten Szenen die größte Informationsmenge und Flexibilität, jedoch auch die höchste Rechenkomplexität. Mit jeder weiteren Verarbeitung von ARD über Data Cubes bis hin zu Composite Layer nimmt das Datenvolumen tendenziell ab, während Homogenität und Datenzugänglichkeit zunehmen. Je besser die Daten vorbereitet und homogenisiert sind, desto einfacher ist der Zugang für raumzeitliche Analysen und der Informationsgewinn

für spezifische Anwendungen. Gleichzeitig wird die Datenverarbeitung zunehmend auf spezifische Anwendungsgebiete zugeschnitten. ARD und Data Cubes sind in der Praxis die gängigsten Modelle für die Speicherung und Analyse raumzeitlicher Rasterdaten und werden im Folgenden ausführlicher erläutert.

3.1.1 ARD

Datenanalysten verbringen rund 80% ihrer Zeit mit der Datenbereinigung, um die Interoperabilität von Zeitreihen sicherzustellen (OGC, 2024, S. 16.). Darüber hinaus ist ein Großteil der Rechenleistung, die für die Speicherung und Verarbeitung dieser Daten erforderlich ist, für Nicht-Fachleute unzugänglich. Eine Lösung hierfür bietet das ARD-Konzept. ARD werden vom Committee on Earth Observation Satellites (CEOS) als Rasterdaten definiert, die nach einem Mindestmaß an Anforderungen verarbeitet und in eine Form gebracht wurden, die zur Unterstützung von Zeitreihenanalysen und der Dateninteroperabilität erforderlich ist (CEOS, n. d., o.S.). Dadurch werden ARD ohne weiteren Aufwand analysierbar und unterstützen die FAIR-Prinzipien, die das Auffinden, den Zugang, die Interoperabilität und Wiederverwendung von wissenschaftlichen Daten fördern (OGC, 2024, S. 13). Das Open Geospatial Consortium (OGC) erarbeitet derzeit eine klare Definition für ARD und nennt als zentrale Eigenschaften eine einheitliche Organisation, umfassende Metadaten, Georeferenzierung und konsistente Zeitreihen. ARD minimieren die Notwendigkeit detaillierter missions- oder sensorspezifischer Kenntnisse und garantieren eine standardisierte Struktur für spezifische thematische Anwendungen. Die OGC stellt folgende Anforderungen an ARD (OGC, 2024, S. 12f.):

- **Datenqualität:** Daten müssen genau, konsistent, vollständig und fehlerfrei sein, wobei Duplikate, fehlende Werte und irrelevante Variablen entfernt werden.
- **Standardisierung:** Einheitliche Datenformate verhindern Kodierungsprobleme in der Analyse.
- **Datenintegration:** Daten aus mehreren Quellen werden zusammengeführt. Eine klare Dokumentation jeder Variablen erleichtert das Verständnis und verbessert die Analysequalität.
- **Datensicherheit und -speicherung:** Sicherer Zugang und Integrität der Daten durch geschützte, zugängliche Speicherung mit Backup- und Wiederherstellungsplänen.

Für Luft- und Satellitenbilder gehören zu den Vorverarbeitungsschritten die radiometrische und geometrische Korrektur, Mosaikbildung, Resampling, Filter zur Wolken- und Schattenmaskierung, Tiling, um parallele Verarbeitung zu erlauben und die Entfernung von Überlappung (Kopp et al., 2019, S. 4ff.). Das CEOS bietet einen umfassenden Überblick über ARD-Datensätze, die viele Satellitenbilder von Missionen wie Sentinel oder Landsat umfassen (CEOS, n. d., o.S.). ARD bietet hierbei optimierte Metadaten zur verbesserten Auffindbarkeit und Nutzung (OGC, 2024, S. 13). Das Konzept fördert die Interoperabilität und trägt dazu bei, dass die Daten den Bedürfnissen der Nutzer und Nutzerinnen gerecht werden. Auch wenn ARD viele Vorteile bringt, bleibt die Umsetzung herausfordernd. Unvollständige Standards und Lücken in den Metadaten erschweren die Konsistenz und Integration. Zudem fehlt eine Abdeckung von Nicht-Earth-Observation-Daten (Non-EO). Darüber hinaus erfordern unterschiedliche Analysebedürfnisse eine flexiblere Datenaufbereitung. Die kontinuierliche Optimierung von ARD zielt darauf ab, die Datenqualität und -repräsentation, standardisierte Maßeinheiten, eine bessere Handhabung der zeitlichen Dimension und eine reichhaltigere Metadatenstruktur zu etablieren. ARD ermöglicht durch einheitliche Standards die vergleichbare Analyse von Bildern mehrerer Sensoren. Dennoch können Herausforderungen wie ungültige Werte, Wolkentrübungen und zeitliche Inkonsistenzen in großen Bildsammlungen auftreten, die zusätzliche Bearbeitungsschritte erfordern, um eine einheitliche Zeitlinie und umfassende Vergleichbarkeit zu gewährleisten (Simoes et al., 2021, S. 3).

3.1.2 Data Cube

Data Cubes, auch als Datenwürfel bezeichnet, sind aus dem Online Analytical Processing (OLAP) von Geschäfts- und Statistikdaten bekannt. In jüngerer Zeit haben Data Cubes auch im Bereich der Geoinformation mehr Aufmerksamkeit erlangt. Dabei handelt es sich um mehrdimensionale Datenstrukturen, die

verschiedene Dimensionen wie Zeit, Raum und Spektralbänder in eine einheitliche Struktur integrieren. Sie eignen sich besonders gut für Zeitreihenanalysen und ermöglichen die Kombination unterschiedlicher Datenquellen, einschließlich Raster- und Vektordaten (Giuliani et al., 2017, S. 102). Während ARD auf Level 2 der Datenprozessierung eingeordnet werden, gehen Data Cubes noch einen Schritt weiter. Sie gelten oft als Level-3-Daten oder „Computation Ready Data“ (CRD), da sie eine nahtlose Integration von Daten über Zeit-, Raum- und Spektraldimensionen ermöglichen (Xu, Du, Fan et al., 2022, S. 1428). Bislang existiert keine einheitliche Definition eines Data Cubes. Giuliani et al., 2017 definiert einen Data Cube als „massives mehrdimensionales Array“, das Daten in einer konsistenten Struktur entlang der d-dimensionalen Achsen speichert und dabei die Speicherressourcen herkömmlicher Serverkapazitäten übersteigt (Giuliani et al., 2017, S. 102). Derzeit wird der Begriff Data Cube oft in Verbindung mit Satellitenbildern verwendet. Ein sogenannter Earth Observation (EO) Data Cube ist eine mehrdimensionale, organisierte Sammlung von Satellitenbildern, die zur Unterstützung von Zeitreihenanalysen verwendet wird (Simoes et al., 2021, S. 2f.). Restriktivere Definitionen erwarten eine einheitliche Projektion und konsistente Zeitachsen. Simoes et al., 2021 konzipieren einen Data Cube als geografisches Feld mit folgenden Merkmalen (Simoes et al., 2021, S. 2f.):

- Jeder Punkt besitzt eine eindeutige Feldfunktion,
- eine zweidimensionale räumliche Komponente,
- eine zeitliche Komponente,
- eine konsistente Attributmenge für alle raumzeitlichen Positionen und
- keine Lücken oder fehlenden Werte.

Es gibt verschiedene Ansätze zur Implementierung des Data-Cube-Datenmodells. Eine Möglichkeit besteht in der Nutzung spezialisierter Software wie Open Data Cube (ODC) oder Pangeo zur Vorverarbeitung, Speicherung, Analyse und Visualisierung. Alternativ bieten Bibliotheken wie *gdalcubes* oder *dtwSat* direkten Zugriff auf raumzeitliche Rasterdaten, während cloudbasierte Dienste wie die Google Earth Engine (GEE) und die Copernicus Data and Information Access Services die Verarbeitung und Analyse großer Data Cubes ermöglichen (Giuliani et al., 2020, S. 1). In den jüngsten Initiativen zur Erstellung von Data Cubes aus Fernerkundungsbildern gilt Australien, insbesondere für die Speicherung und Verarbeitung raumzeitlicher Rasterdaten im Kontext der Zeitreihenanalyse, als Vorreiter. Weitere Initiativen wie der Data Cube der Schweiz, der armenische Data Cube und der afrikanische Data Cube folgten diesem Beispiel. Ein Data Cube verringert die Anzahl der erforderlichen Zugriffe auf Einzeldateien und optimiert zeitliche Abfragen im Vergleich zur herkömmlichen Speicherung einzelner Szenen oder ARD. Bei der Erstellung eines Spektralprofils beispielsweise ermöglicht der Data Cube einen erheblichen Zeitgewinn, da er im Gegensatz zu einzeldateibasierten Systemen weniger Zugriffe erfordert, da diese primär für räumliche Abfragen optimiert sind (Kopp et al., 2019, S. 8). Data Cubes ermöglichen es, den Schwerpunkt stärker auf die Datenverarbeitung zu legen, anstatt sich auf Speicherverwaltung, Rechenkapazitäten und Kosten zu konzentrieren. Data Cubes erfordern jedoch erhebliche Speicherressourcen. Anwendungen mit mittlerer und hoher Auflösung führen zu spärlichen Data Cubes und benötigen eine umfassende Vorverarbeitung, um eine konsistente und vollständige Datenbasis zu gewährleisten. Ferner sind sie auf klare Standards und Vorverarbeitungsmethoden angewiesen. Um die Bildung von Data Cubes weiter zu unterstützen, wurden in den letzten Jahren neue Initiativen ergriffen. Insbesondere das CEOS und die Open Data Cube Initiative sind Vorreiter in diesem Gebiet und versuchen mit neuen Technologien, Software und Workflows den Zugriff auf diese enormen Datenmengen zu vereinfachen (Ferreira et al., 2020, S. 1). Im Jahr 2019 gab es weltweit etwa 9 Data Cubes für Erdbeobachtungsdaten, die die Open Data Cube Technologie nutzen (Kopp et al., 2019, S. 2).

3.2 Speicherung raumzeitlicher Rasterdaten

Das schnelle Wachstum von raumzeitlichen Daten hat viel Aufmerksamkeit auf sich gezogen. Mit dem Start von Landsat 9 im Jahr 2021 beobachtet die Landsat-Satellitenserie die Erde seit fast 50 Jahren kontinuierlich. Die Sentinel-Satelliten der Europäischen Weltraumorganisation (ESA) hatten bis Ende 2020

24,87 Petabyte an Fernerkundungsdaten erhalten (Xu, Du, Fan et al., 2022, S. 1418). Der dramatische Anstieg des Datenvolumens löst Herausforderungen in der technischen Umsetzung aus, da das Datenvolumen die Kapazität herkömmlicher Computertechnologien bei weitem übersteigt. Infolgedessen haben nur wenige Forschungseinrichtungen und Unternehmen Zugang zu diesen großen raumzeitlichen Rasterdaten, was zu Schwierigkeiten bei der Nutzung und Limitierungen der Entwicklung führt (Xu, Du, Fan et al., 2022, S. 1418).

Cloud Computing ist ein Big-Data-Dienst, der über das Internet bereitgestellt wird und in den letzten Jahren häufiger für große raumzeitliche Datenmengen genutzt wird. Die Nutzung bietet sich an, da Cloud Computing in der Regel als „Pay-per-use“ angeboten wird und die elastische Ausdehnung von Ressourcen auf Abruf unterstützt (Xu, Du, Fan et al., 2022, S. 1418). Infolgedessen sind die Kosten viel niedriger als bei anspruchsvollen und teuren Hochleistungscomputern. Außerdem wird Cloud Computing über das Internet bereitgestellt, was den offenen Austausch von Rasterdaten ermöglicht und somit den Fortschritt der FAIR-Prinzipien (auffindbar, zugänglich, interoperabel und wiederverwendbar) fördert. Dadurch hilft Cloud Computing sich mehr auf Algorithmen und datensintensive Analysen zu konzentrieren, anstatt durch die Computertechnologie eingeschränkt zu werden.

Konventionell werden Rasterdaten als Arrays in mehreren Dateiformaten gespeichert, einschließlich des hierarchischen Datenformats (HDF), GeoTIFF (Landsat) und JPG2000 (Sentinel-2) (Xu, Du, Fan et al., 2022, S. 1418). Die Größe eines einzelnen Datensatzes reicht im Allgemeinen von Megabyte bis Terabyte. Darüber hinaus speichern einige Cloud-basierte Plattformen Rasterdaten als Kacheln in leichten Datenformaten für schnellere Visualisierung, wie z.B. PNG und JPEG. Konventionelle Dateisysteme haben jedoch Schwierigkeiten, mit den enormen Datenmengen umzugehen, weil diese Daten das Speichervolumen herkömmlicher Speicherhardware, wie Festplatten, bei weitem übersteigen. Verteilte Speichersysteme können zwar große Mengen an Daten speichern, aber die Kosten für den Speicher sind sehr hoch, was für Einzelpersonen oder Unternehmen eine finanzielle Belastung darstellt. Zum Beispiel erwog der United States Geological Survey (USGS) 2018 eine Gebühr für den Zugang zu weit verbreiteten Quellen für Fernerkundungsdaten wie z. B. Landsat einzuführen, um die Kosten zu decken (Xu, Du, Fan et al., 2022, S. 1419). Ein weiteres Problem ist, dass der Zugriff auf diese Daten oft ineffizient ist. Bei Analysen, wie Zeitreihenanalysen, sind die Daten häufig über viele Dateien verstreut, was viele zufällige Zugriffe und aufwendige Datenumwandlungen erfordert. Konventionelle Big-Data-Technologien sind für solche Anforderungen oft nicht optimiert, weshalb neue Speicherlösungen entwickelt werden müssen, um diese spezifischen Herausforderungen zu bewältigen. Aktuell gibt es fünf führende Cloud Computing- und Big-Data-Technologien, die für diesen Zweck geeignet sind (Xu, Du, Fan et al., 2022, S. 1423 f.):

- **Object Storage Systems (OSS):** OSS verwalten Daten als Objekte, die jeweils durch eine global eindeutige Kennung identifiziert werden. Der Zugriff erfolgt über HTTP mittels RESTful-APIs. Sie sind besonders gut geeignet, um große Mengen an Daten in der Cloud zu speichern und bieten hohe Flexibilität durch einfache Integration in Netzwerke. Allerdings unterstützt OSS keine traditionellen Dateioperationen wie Öffnen, Lesen oder Schreiben, was die direkte Verarbeitung erschwert. Stattdessen muss eine Datei vollständig heruntergeladen, bearbeitet und wieder hochgeladen werden.

- **Distributed File Systems (DFS):** DFS wie das Google File System oder das Hadoop Distributed File System bieten umfangreichere Schnittstellen und Funktionen als OSS. Sie unterstützen komplexe Dateioperationen, einschließlich Anhängen und Änderungen, was für die Verarbeitung von Rasterdaten vorteilhaft ist. Beispielsweise speichert Digital Earth Australia Landsat-Archive im Lustre-System. Allerdings können DFS unter Skalierungsproblemen leiden, insbesondere wenn zentrale Knotenpunkte Engpässe verursachen.

- **Relationales Datenbankmanagementsystem (RDBMS):** RDBMS wie PostgreSQL sind zuverlässige Tools zur Speicherung und Verwaltung von Metadaten, die transaktionale Operationen und

ACID-Eigenschaften unterstützen. Erweiterungen wie PostGIS Raster ermöglichen auch die Speicherung von Rasterdaten, wodurch komplexe Analysen mit SQL möglich werden. Allerdings stoßen sie bei der Handhabung sehr großer Datenmengen an ihre Grenzen, was ihre Eignung für massive Rasterdatensätze einschränkt (MapScaping Aps, 2023, o.S.). Rasterdaten sind sehr umfangreich, was die Verwaltung und Skalierbarkeit der Datenbank erschwert. RDBMS sind für Terabyte-Datenmengen weniger geeignet, da Backup, Replikation und Speicherbedarf problematisch werden. Darüber hinaus nutzen Rasterdaten viele der Stärken einer relationalen Datenbank nicht optimal. Datenbanken sind darauf ausgelegt, häufig geänderte Daten zu verwalten und für Ausfallsicherheit zu sorgen, aber Rasterdaten werden selten modifiziert. Außerdem sind Rasterdaten bereits in speziellen Strukturen organisiert, die schnellen Zugriff ermöglichen, wodurch die Integration in eine Datenbank keinen signifikanten Geschwindigkeitsvorteil bringt. Während die Rasterdatenverarbeitung von Natur aus parallel ist, passt sie nicht gut zu herkömmlichen Datenbankmodellen. SQL-basierte Systeme arbeiten in der Regel zeilenweise und haben das detaillierte Wissen über die Datenstruktur, die Raster-native Frameworks besitzen. Und obwohl die Architektur von PostGIS einige Parallelitäten ermöglicht, kann sie die Arbeitslast bisher nicht über eine große Anzahl von Kernen verteilen (MapScaping Aps, 2023, o.S.).

- **NoSQL-Datenbanken:** Mit dem Aufkommen von Web 2.0 und der Verwaltung großer Mengen unstrukturierter Daten haben sich NoSQL-Datenbanken wie MongoDB, HBase und Google Big Table etabliert. Sie betonen Konsistenz, Verfügbarkeit und Partitionstoleranz (CAP), bieten hohe Skalierbarkeit und Effizienz, unterstützen jedoch keine ACID-Eigenschaften oder räumliche Indizes, der für raumzeitliche Rasterdaten und raumzeitliche Abfragen benötigt werden. Array-Datenbankmanagementsysteme (DBMS), wie SciDB oder TileDB, werden oft unter NoSQL eingeordnet und sind speziell für wissenschaftliche Array-Daten ausgelegt. Sie ermöglichen SQL-ähnliche Abfragen und unterstützen Array-Operationen wie Resampling und Aggregation. Ihre optimierten I/O-Technologien und Unterstützung für horizontale Skalierung machen sie besonders geeignet für Cloud Computing. Beispiele wie TileDB nutzen Shared-Nothing-Architekturen, die Cloud-Objektspeicher wie AWS S3 integrieren, während Systeme wie RasDaMan in Projekten wie EarthServer Fernerkundungsdaten effizient verwalten. Die Speicherung in Array-DBMS ist jedoch zeitaufwändig und meist kostspieliger als OSS- oder DFS-Lösungen.

Die meisten dieser Möglichkeiten lassen sich als IaaS-Bereitstellungsmodell realisieren. Darüber hinaus bieten Cloud-Dienste auch Speicherdienste als SaaS an, wodurch die aufwendige Datenbankwartung vermieden wird. Cloud-optimierte Datenformate werden optimiert, um die E/A-Leistung im Cloud-Speicher zu verbessern. OSS unterstützt keine Dateiöffnungs- und Schreibvorgänge, was für raumzeitliche Analysen unpraktisch ist. Auch der partielle Zugriff auf einen kleinen Teil der Daten wird nicht nativ unterstützt. Stattdessen muss der gesamte Fernerkundungsdatensatz heruntergeladen werden, was zu hohen redundanten Gemeinkosten führt. Cloud-optimierte Datenspeicherformate wie Zarr und COG haben sich etabliert und haben die Leistung der Rasterdatenspeicherung verbessert. Diese Formate ermöglichen es auf einen Teil der Rasterdaten zuzugreifen, ohne den gesamten Datensatz herunterladen zu müssen. Infolgedessen kann COG die Abrufeffizienz sowohl in DFS als auch in OSS verbessern. So entwickelte sich COG im Jahr 2021 als Standarddatenformat für die Datensammlung der Landsat-Serie 2 (Xu, Du, Fan et al., 2022, S. 1425 f.). Benutzer können mit geringen Ladezeiten und ohne Berücksichtigung von Datenmanagement und Serverwartung effizient auf die Daten zugreifen, was die FAIR-Prinzipien stark fördert. Eine Alternative ist die Verwendung der cloud-optimierten Rasterdatenformate bzw. das Speichern von Verweisen auf diese in einem RDBMS (Out-DB Rasters) (MapScaping Aps, 2023, o.S.). Diese Ansätze bieten flexiblere und effizientere Möglichkeiten, um große Rasterdatensätze mit einer Datenbank zu verwalten, ohne die Komplexität einer Datenbankintegration. Da sich die Daten jedoch nicht lokal in der Computerumgebung befinden, kann der Zugriff auf sie mit einer gewissen Latenz verbunden sein. Diese Latenz kann noch ausgeprägter sein, wenn sich die Datenbank und die Geodaten nicht in derselben Cloud befinden. Aufgrund dessen werden zur Zeit vorallem Lösungen mit cloud-optimierten Datenformaten in OSS angestrebt.

3.3 Cloud-native Geospatial

Die rasante Entwicklung von Cloud-Technologien in den letzten Jahren hat die Disziplin der Geo-Wissenschaft nicht unberührt gelassen. Paradigmen wie „mehr Zugang, mehr Nachfrage“ sowie Globalisierungsaspekte machen Geodaten und -technologien abhängiger denn je von der Cloud (CyberSWIFT, 2022, o.S.). Mit der wachsenden Nutzerzahl, die Zugang zu diesen stetig wachsenden Datenmengen sucht, entstehen vermehrt cloud-native Ansätze. Allerdings werden die Vorteile des Cloud Computing für Geodaten oftmals nicht voll ausgeschöpft. Cloud-native Geospatial-Technologien spielen eine wichtige Rolle, um das Potenzial des Cloud Computing für Geodaten weiter zu erschließen. Cloud-native Geospatial umfasst Standards und Software, die von Grund auf für die Cloud entwickelt wurden. Dabei liegt der Fokus darauf, die Vorteile der Cloud-Infrastruktur bei der Erstellung von Geoanwendungen und -diensten optimal auszunutzen (Di und Sun, 2023, S. 126). Cloud-native Geospatial-Technologien ermöglichen es, Geodaten noch effizienter in der Cloud zu verwalten, zu analysieren und zu visualisieren, was zu einer Verbesserung der Entscheidungsfindung und Reaktion auf geowissenschaftliche Herausforderungen führen kann. Der Fokus von cloud-nativ Geospatial liegt dabei aktuell auf dem cloudbasierten Zugriff (Holmes, 2021, o.S.).

Traditionell erfolgt der Zugriff auf Geodaten im sogenannten Data-to-Code-Modell bzw. Download-Modell. Dabei werden die Daten von einem zentralen Server heruntergeladen und lokal verarbeitet (siehe Abbildung 10). Dieser Ansatz ist zeitaufwändig, wenig reproduzierbar und erfordert manuelle Identifizierung, Download und Verarbeitung. Bei großen Datensätzen ist er ineffizient, da paralleles oder verteiltes Computing notwendig wird (R. P. Abernathey et al., 2021, S. 26). Um diese Herausforderungen zu überwinden, setzen cloud-native Ansätze auf das Code-to-Data-Modell, bei dem die Verarbeitung näher an der Datenquelle in der Cloud stattfindet (siehe Abbildung 11). Dieser Ansatz nutzt Objektspeicher wie Amazon S3, Google Cloud Storage oder Azure Blob Storage, um hohe Flexibilität und Skalierbarkeit zu erzielen. Geodatenanalysesysteme verwenden häufig noch dateibasierte Zugriffe über POSIX-Dateisysteme, die auf Dateien und Verzeichnissen basieren. Diese sind jedoch für lokale Umgebungen optimiert und arbeiten weniger effizient in Cloud-Umgebungen, da sie nicht auf die hohe Latenz und den hohen Durchsatz von Objektspeichern abgestimmt sind. Dies führt zu Leistungseinbußen bei der Datenverarbeitung in der Cloud mit herkömmlichen Dateiformaten (R. P. Abernathey et al., 2021, S. 29f.). Cloud-optimierte Formate hingegen sind besser an die Eigenschaften von Objektspeichern angepasst und bieten dadurch eine deutlich verbesserte Leistung und Skalierbarkeit. Sie sind speziell für die effiziente Speicherung und Übertragung großer Datensätze in der Cloud konzipiert und ermöglichen schnellen Zugriff sowie effizientes Rendering.

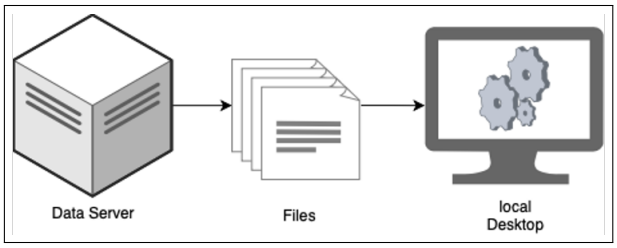

Abbildung 10: Data to Code

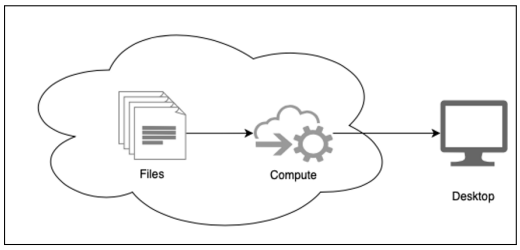

Abbildung 11: Code to Data

Während herkömmliche Geodatenbanken wie PostGIS weiterhin für robustes Datenmanagement bei hoher Schreib- und Updatelast geschätzt werden, bieten cloud-native Ansätze zusätzliche Vorteile. Neben Performance- und Skalierungsgründen bietet cloud-native Geospatial viele Vorteile, wie die Verringerung der Belastung der Datenanbieter und die Senkung der Kosten für die Verwaltung von Geodaten. Durch den Einsatz von cloud-optimierten Datenformaten, wie GeoParquet oder COG, können Geodaten effizienter in der Cloud gespeichert, übertragen und verarbeitet werden. Cloud-optimierte Datenformate sind speziell dafür konzipiert, sich nahtlos in die Cloud-Infrastruktur zu integrieren. Sie ermöglichen es, nur die benötigten Teile eines Datensatzes zu laden, was zu erheblichen Einsparungen bei den Kosten für Datenübertragung und Laufzeit führt. Darüber hinaus unterstützen sie parallele Verarbeitung und

einfache Datenbereitstellung, wodurch sie ideal für riesige Datensätze sind. Die Standardisierung von Datenformaten und -protokollen fördert den nahtlosen Datenaustausch und die Integration, was für eine reibungslose Interoperabilität zwischen verschiedenen Systemen und Plattformen entscheidend ist. Diese Bemühungen werden durch die FAIR-Prinzipien unterstützt, die 2016 mit dem Artikel von Wilkinson aufkamen (M. Wilkinson et al., 2016, o.S.). Die Prinzipien von FAIR - Findability, Accessibility, Interoperability und Reusability - legen fest, dass Daten eine eindeutige Kennung und Metadaten haben sollten, um sie leicht auffindbar zu machen, über ein standardisiertes Protokoll zugänglich sein müssen, interoperabel sein sollen und so aufbereitet sein müssen, dass sie wiederverwendet werden können. Weltweit engagieren sich zahlreiche Initiativen wie die GOFair-Initiative und Force11 für die Umsetzung der FAIR-Prinzipien in wissenschaftlichen Daten. Verschiedene Fachgebiete entwickeln eigene Standards auf Basis dieser Prinzipien, darunter das OGC. Die OGC engagiert sich aktiv in der Förderung und Standardisierung von cloud-optimierten Geodatenformaten als Teil ihrer Mission, um die Interoperabilität zu fördern (Open Geospatial Consortium, n. d., o.S.). Im Fokus steht die Entwicklung von cloud-opimierten Datenformaten, die eine Streaming-Anzeige und „on the fly"-Analyse ermöglichen (Holmes, 2021). Darüber hinaus wird ein einheitliches Metadatenformat benötigt, welches verschiedenste Geodatenformate für Raster- und Vektordaten, Punktwolken, Videos sowie 3D-Animationen unterstützt sowie spezifische georäumlich Attributbeschreibungen bereitstellt. Besonders COG hat sich in zahlreichen Anwendungsfeldern im Bereich der Rasterdaten etabliert. Analog zu COG wurde für Punktwolken das Format Cloud-Optimized Point Clouds (COPC) entworfen. Für mehrdimensionale Arrays bietet sich Zarr als cloud-optimiertes Format an (Maskey et al., 2023, S. 209). Auch TileDB, COMTiles und PMTiles erfüllen die Anforderungen für cloud-optimierte Datenformate. Für Vektordaten kommen zunehmend Formate wie FlatGeobuf und GeoParquet zum Einsatz (siehe Tabelle 1).

Kategorie	**Format**
Raster	Cloud Optimized GeoTiff (COG)
Point Clouds	Cloud Optimized Point-Clouds (COGC)
N-dimensional Array	Zarr, Kerchunk
Vektor	FlatGeobuf, Geoparquet
Tile Archiv	PMTiles, COMTiles

Tabelle 1: Cloud-optimierte Datenformate

Solche cloud-optimierten Geodatenformate bieten effiziente Lösungen für den Umgang mit großen Geodatensätzen. Sie ermöglichen eine direkte Verarbeitung der Daten aus dem Objektspeicher ohne lokale Speicherung und liefern Vorteile wie hohe Verfügbarkeit, skalierbare Leistung und geringe Latenz. Zentrale Eigenschaften dieser cloud-optimierten Formate sind dabei (R. P. Abernathey et al., 2021, S. 30):

- ***Metadaten:***Ausreichende Metadaten und persistente Identifikatoren für eine einfache Bereitstellung und Zugriff auf Informationen.
- ***Read-Orientierung:*** Nutzung des HTTP-Protokolls für den effizienten Zugriff ohne Dateipfade.
- ***Partitionierung:*** Die Daten sind in interne Gruppierungen wie z. B. Chunks oder Tiles organisiert, die parallele und verteilte Verarbeitung sowie einen partiellen Zugriff auf Teilmengen ermöglichen.
- ***Interoperabilität:*** Maximale Interoperabilität zwischen verschiedenen Diensten und Katalogen.
- ***Open-Source:*** Open-Source-Natur für Flexibilität und Anpassbarkeit an individuelle Bedürfnisse.
- ***Datenkompression:*** Datenkompression zur Reduzierung des Speicherbedarfs und der Kosten.

Die Struktur cloud-optimierter Formate folgt in der Regel einem bestimmten Aufbau, der aus verschiedenen Komponenten besteht: dem Header, Metadaten, einem Index und den eigentlichen Daten. Der

Header und die Metadaten enthalten Informationen über die Datei und ihre Inhalte. Indexierungen ermöglichen einen schnellen und effizienten Zugriff auf spezifische Teile der Daten. Die eigentlichen Geodatensätze wie Satellitenbilder, Punktwolken oder Vektorgeometrien sind in der Datenkomponente enthalten. Cloud-optimierte Formate erlauben es durch diese Merkmale, den Fokus von Datenmanagement und -pflege hin zu Datenanalyse und -nutzung zu verschieben, was letztendlich zu Kosteneffizienz und erhöhter Agilität führt. Dies ist vor allem im Hinblick auf die offene Bereitstellung von ARD entscheidend. ARD sind sofort für die weitere Verarbeitung und Visualisierung nutzbar, was die Geschwindigkeit der Datenanalyse und die Ableitung von Erkenntnissen erheblich beschleunigt (R. P. Abernathey et al., 2021, S. 28f.). Cloud-optimierte Datenformate bringen durch die geringere Latenz bei Datenabfragen und die Interoperbailität einen nahtlosen Datenaustausch. Durch die Unterstützung der Skalierbarkeit und Parallelverarbeitung bringen diese Formate erhebliche Vorteile in einem cloud-basierten Umfeld.

3.4 Cloud-optimierte Rasterdatenformate

In diesem Abschnitt werden die cloud-optimierten Rasterdatenformate COG und Zarr ausführlich beschrieben, die im vorherigen Kapitel im Zusammenhang mit Rasterdaten erwähnt wurden.

3.4.1 COG

Traditionell werden hochauflösende Rasterbilder von Servern heruntergeladen, um diese analysieren oder visualisieren zu können. Dies kann unter Umständen eine beträchtliche Downloadzeit erfordern. Insbesondere im Hinblick auf die Erstellung von reaktionsschnellen Echtzeitanwendungen werden mittlerweile vermehrt Cloud-Dienste mit nahezu unbegrenztem Speicherplatz genutzt, deren Kosten von der Nutzungsintensität abhängen (Open Geospatial Consortium, 2022a, o.S.). Das cloud-optimierte GeoTIFF (COG) ermöglicht effizientere Workflows in der Cloud-Umgebung. Die Grundlage für das COG bildet das TIFF-Format. TIFF, kurz für „Tag Image File Format“, ist ein Dateiformat, das für die Speicherung von Rastergrafikbildern entwickelt wurde. Es dient als Standardformat, das eine breite Akzeptanz und Kompatibilität für gescannte Bilddateien sowie Kompressionsmechanismen gewährleistet. Mit dem Geography Tagged Image File Format (GeoTIFF) wurden zusätzliche Tags für geographische Metadaten als Erweiterung des ursprünglichen Standards eingeführt. Dies umfasst beispielsweise die Angabe eines Koordinatensystems sowie die Auflösung in x- und y-Richtung. Der GeoTIFF-Standard hat sich mittlerweile im Bereich der Geoinformation und Fernekrundung etabliert. COG beruht daher auf dem TIFF- und GeoTIFF-Standard und erweitert diese um zwei weitere Kerntechnologien. Die Unterstützung für den neuen Standard wurde schnell in Open-Source-Softwarebibliotheken integriert und behält die Abwärtskompatibilität bei. Systeme, die eine reguläre TIFF-Datei anzeigen können, können auch eine COG anzeigen, auch wenn sie die georäumlichen Komponenten eines GeoTIFFS oder die Technologie der HTTP-Ranges nicht nutzen.

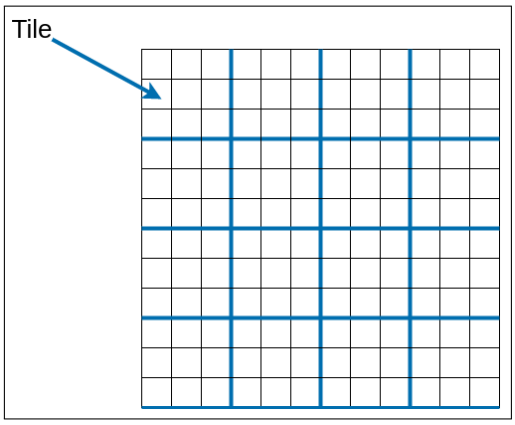

Abbildung 12: Tiling einer COG-Datei (Planet Labs PBC, 2024, o.S.)

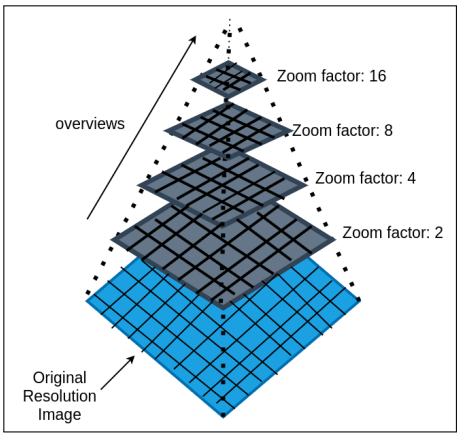

Abbildung 13: Overviews einer COG-Datei (Planet Labs PBC, 2024, o.S.)

Zum Einen gehört zu den Kerntechnologien von COG die Fähigkeit, Pixel durch Tiling und Overviews zu organisieren. Beim Tiling werden für jedes Rasterbild Kacheln erzeugt. Dabei ist das Gesamtbild in kleinere, rechteckige Tiles aufgeteilt, die jeweils ein festes, räumlich begrenztes Gebiet abdecken. Die Größe der Kacheln kann je nach Anwendungsbedarf angepasst werden, wobei übliche Größen 512x512 Pixel betragen (Cloud-Native Geospatial Foundation, 2023, o.S.). Im Gegensatz zu der üblichen Methode, Rasterbilder in Streifen zu speichern, ermöglicht das Kacheln einen schnelleren Zugriff auf einen Teilbereich der Datei, da nicht die gesamte Datei gelesen werden muss (siehe Abbildung 12). Dies ist besonders nützlich, wenn ein kleiner Teil der gesamten Datei verarbeitet oder visualisiert werden muss. Die Kacheln organisieren alle relevanten Bytes eines Bereichs innerhalb derselben Dateiabschnitte, wodurch Range-Requests nur die benötigten Daten abrufen können. Die Overviews sind verkleinerte Versionen desselben Rasterbildes mit weniger Details und kleinerer Größe. Diese erhöhen zwar die Gesamtdateigröße, können jedoch viel schneller bereitgestellt werden, da der Renderer nur die Werte in der Übersicht zurückgeben muss, anstatt diese zu berechnen. Oft hat ein einzelnes GeoTIFF viele Übersichten, um verschiedenen Zoomstufen zu entsprechen und die Leistung zu verbessern (Open Geospatial Consortium, 2022a, o.S.). Overviews werden genutzt, wenn ein schnelles Bild der gesamten Datei gerendert werden soll. Dabei muss nicht jedes Pixel heruntergeladen werden, sondern es wird lediglich die viel kleinere, bereits erstellte Übersicht angefordert (siehe Abbildung 13).

Zum Anderen ermöglicht COG die Nutzung von HTTP-GET-Range-Requests, mit denen Teile einer Datei angefordert werden können. Dieses Verfahren wird Byte-Serving genannt. Dabei wird zuerst der Header der Datei übertragen, der wichtige Metadaten enthält, wie die Position der einzelnen Teile der TIFF-Datei und die GeoTIFF-Tags für die räumliche Orientierung. Mit diesen Informationen kann der Client genau bestimmen, welche Datenbereiche er für die Anzeige oder Verarbeitung benötigt. Durch weitere HTTP-Range-Requests werden dann nur diese relevanten Teile der Datei übertragen. Da der Range-Header jedoch optional ist, kann weiterhin, wenn nötig, auf die gesamte Datei zugegriffen werden. Innerhalb der COG-Datei werden die Kacheln dafür in einer speziellen Reihenfolge gespeichert, die auf „Byte-Serving“ optimiert ist. Zur Verwaltung der Kacheln, sowie der Beziehung der einzelnen Kacheln werden Image File Directories (IFDs) verwendet. Ein IFD enthält eine Liste von Referenzen in Form von Key-Value-Paaren, die auf das tatsächliche Rasterbild sowie auf zusätzliche Metadaten verweisen. Die IFDs sind wichtig, da sie Informationen über die Kachel-Offsets und die Kachel-Byteanzahl enthalten. Bei einer typischen COG-Datei gibt es ein IFD für die Rohdaten in voller Auflösung und zusätzliche IFDs für jeden Overview (siehe Abbildung 14). Teile der Datei können so effizient abgerufen werden, da schnell bestimmt werden kann, wo sich die einzelnen Kacheln in der Datei befinden Planet Labs PBC, 2024. Diese Struktur ermöglicht partielle Zugriffe und parallele Lesevorgänge.

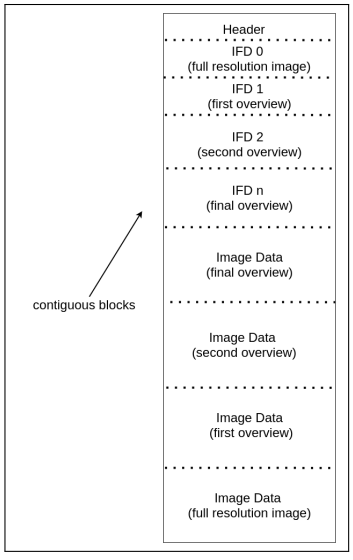

Abbildung 14: Aufbau einer COG-Datei (Planet Labs PBC, 2024, o.S.)

3.4.2 Zarr

Zarr ist ein Gemeinschaftsprojekt von der US-amerikanischen gemeinnützigen Organisation NumFOCUS zur Entwicklung von Spezifikationen und Software zur Speicherung von chunked, komprimierten, N-dimensionalen Arrays. Es wurde entwickelt, um den Bedarf an einem einfachen, transparenten, offenen und gemeinschaftsgetriebenen Format zu decken, das große Datenmengen effizient auf verteilten Speichersystemen speichern und lesen kann. Dies ist insbesondere für Anwendungen mit hohem Datendurchsatz und großen Datensätzen von Interesse (Zarr, 2024, o.S.). Zarr wird bislang primär im Bereich der Klimatologie und Ozeanographie eingesetzt, findet aber immer mehr Anwendung im Kontext von Satellitenbildern. Zarr ist daher zwar kein inhärentes Geodatenformat, aber aufgrund seines schnellen Wachstums und seiner Akzeptanz im Geoinformationsbereich wird es von der OGC als OGC-Community-Standard unterstützt (Open Geospatial Consortium, 2022c, S.3). Die Zarr-V2-Spezifikation enthält daher zunächst keinen Verweis auf Geodatenkonzepte. Es haben sich jedoch einige gemeinsame Praktiken und eine GeoZarr-Spezifikation in Bezug auf Geodaten herausgebildet, die es beispielsweise erlauben Koordinatenreferenzsystem in den Metadaten der Zarr-Arrays zu speichern (Open Geospatial Consortium, 2022c, S.4ff.). Zarr-Daten können in jedem Speichersystem gespeichert werden, das als Key-Value-Speicher dargestellt werden kann. Dazu gehören in der Regel POSIX-Dateisysteme und Cloud-Objektspeicher, aber auch Zip-Dateien sowie relationale und Dokumentendatenbanken. Häufig dient Zarr deshalb zur Speicherung von HDF5, NetCDF und Rasterdaten (Zarr, 2024, o.S.).

Zarr ist ein flexibles und leistungsfähiges Dateiformat zur Speicherung und zum Zugriff auf große, mehrdimensionale Datenmengen (Zarr, 2024, o.S.). Es basiert auf einem hierarchischen Modell, das von HDF5 inspiriert ist und ermöglicht die Organisation von N-dimensionalen Arrays und Gruppen in einer strukturierten Ordnerhierarchie. Diese Struktur erlaubt die effiziente Verarbeitung und Verwaltung großer Datensätze in verteilten Systemen. Ein Zarr-Datensatz besteht aus einer Hierarchie von Gruppen (vergleichbar mit Ordnern) und Arrays (vergleichbar mit Dateien) (siehe Abbildung 15). In jeder Gruppe (.zGroup) können mehrere Arrays gespeichert werden. Jedes Array verfügt über seine eigenen Metadaten und Attribute (.zarray und .zattrs). Die Metadaten werden dabei getrennt von den Daten gespeichert. Die Metadatenressource ist ein JSON-Objekt und enthält Informationen zum Datentyp, Größe, die Kompressionsmethode oder die Chunking-Strategie (Zarr Developers, 2024, S. 91). Die Größe eines Zarr-Arrays ist das Tupel aus den Längen der Arrays in jeder Dimension. Arrays repräsentieren die eigentlichen Daten und sind in Chunks unterteilt – kleine, unabhängige Blöcke, die individuell gelesen, geschrieben und komprimiert werden können (Zarr Developers, 2024, S. 91). Jeder Chunk ist als eigene Datei gespeichert und die Dateinamen geben dessen Position im N-dimensionalen Datenraum an. Somit entspricht jeder Chunk in Zarr einem bestimmten Bereich innerhalb des multidimensionalen Data Cubes und kann eigenständig adressiert und gelesen werden. Für ein Array mit den Dimensionen x, y und Zeit könnte eine Datei mit dem Namen „2.3.1“ den Chunk bezeichnen, der den dritten Chunk in der x-Dimension, den vierten in der y-Dimension und den zweiten in der Zeit-Dimension enthält (Zählung beginnt bei 0). Diese Struktur ermöglicht eine eindeutige und einfache Identifikation jedes Chunks basierend auf seiner Position im mehrdimensionalen Datenraum. Die Größe der Chunks kann für jede Dimension eines Arrays flexibel konfiguriert werden, um eine optimale Anpassung an die Verarbeitungsanforderungen und Datencharakteristika zu ermöglichen. Wichtig ist, dass alle Chunks eines Arrays innerhalb jeder Dimension dieselbe Größe haben, um die Konsistenz der Datenstruktur zu gewährleisten.

In Abbildung 15 ist das Array in vier Chunks unterteilt, die in einem 2x2 Raster angeordnet sind. Das Array verfügt dementsprechend über zwei Dimensionen. Jeder Chunk wird mit einer eindeutigen ID identifiziert, die beispielsweise als (0.0), (0.1) usw. bezeichnet werden kann. Nach der Chunking-Phase folgt die Filterung und Kompression der Daten. Diese Schritte zielen darauf ab, irrelevante oder redundante Informationen zu entfernen und die Daten auf eine effiziente Weise zu komprimieren, um Speicherplatz zu sparen und den Datenzugriff zu beschleunigen. Anschließend werden die gefilterten und komprimierten Daten als Key-Value-Paare gespeichert. Dabei wird jeder Chunk-ID ein entsprechender komprimierter Datensatz zugeordnet, der als Wert des Key-Value-Paares fungiert (Miles, 2019, o.S.). Der partiellen Zugriff kann somit über eine effiziente Indexierung durch NumPy-ähnliche Slicing-Operationen erfolgen.

Dabei wird zuerst der Header bzw. die Metadaten der Datei übertragen. Die Metadaten des Arrays enthalten Informationen über die Dimensionen des gesamten Datensatzes, die Abmessungen der Chunks und die Gesamtgröße des Datenbereichs. Dadurch ist es möglich, die Position der benötigten Chunks präzise zu berechnen. Bei einer räumlichen Bereichsanfrage wird anhand der Koordinaten des gewünschten Bereichs im Verhältnis zum räumlichen Ausmaß des gesamten Datensatz der Chunk-Index der jeweiligen Dimension x und y ermittelt. Diese Indizes definieren wiederum die Namen der benötigten Chunk-Dateien, die dann gezielt geladen werden können. Wenn in der x-Dimension der zweite Chunk, in der y-Dimension der erste Chunk und alle Zeitpunkte abgefragt werden, müssen alle Chunk-Dateien geladen werden, deren Namen mit „1.0.“ beginnen. Dadurch können nur die relevanten Chunks geladen werden, ohne den gesamten Datensatz einlesen zu müssen.

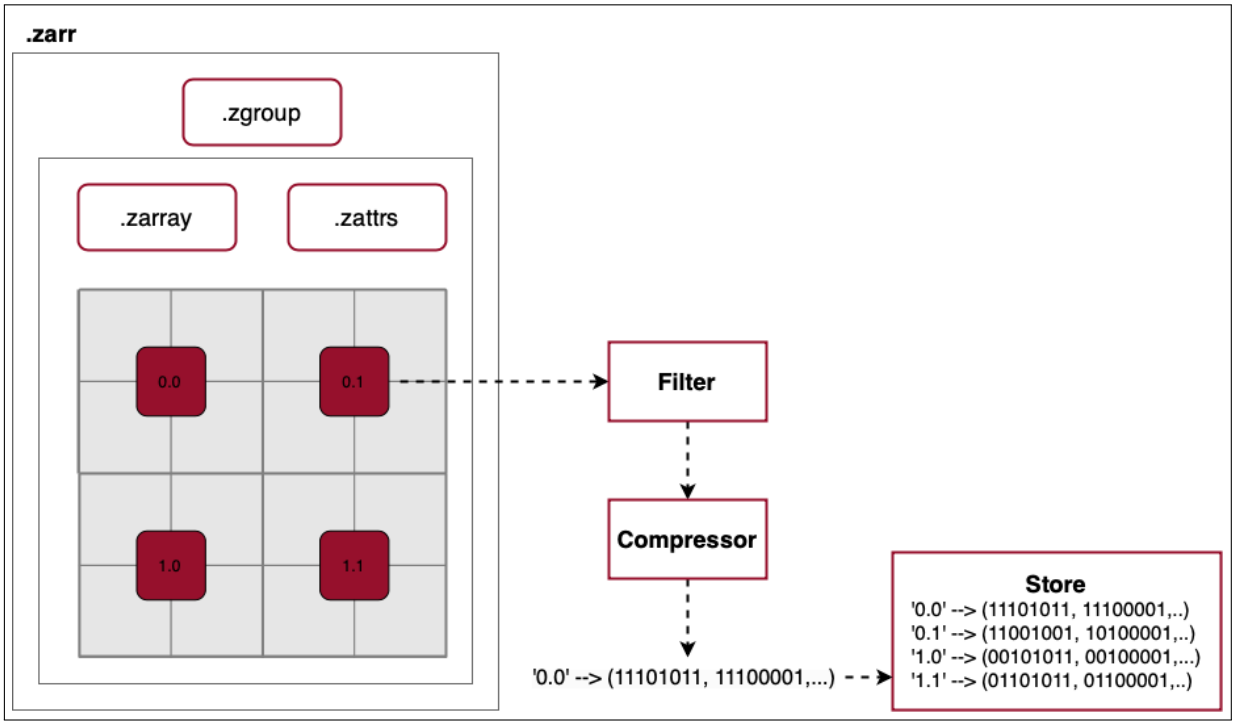

Abbildung 15: Aufbau einer Zarr-Datei (Eigene Darstellung nach Miles, 2019, o.S.)

Wegen der Fähigkeit Datenblöcke abrufen zu können, ist Zarr für Szenarien geeignet, in denen auf bestimmte Teile des Datensatzes zugegriffen wird. Ein weiterer Vorteil von Zarr ist, dass es eine effiziente Möglichkeit zum Hinzufügen, Entfernen und Modifizieren von Datenblöcken bietet, was es zu einer guten Wahl für Datensätze macht, die sich häufig ändern. Weiterhin unterstützt Zarr die parallele Verarbeitung und bietet eine umfangreiche Unterstützung für Komprimierungen sowie verschiedener Speichersysteme. Auch die Zusammenarbeit mit verschiedensten Frameworks wie Dask und Xarray ist mit Zarr möglich (Abdishakur, 2022, o.S.). Diese Leistungsfähigkeit macht Zarr besonders geeignet für Big-Data-Anwendungen. Die Anwendungsbereiche reichen von einfachen und schnellen Serialisierungen von NumPy-Arrays bis zur n-dimensionale Speicherung von georäumlichen Rastern (Zarr, 2024, o.S.). Ein Beispiel ist das CMIP6-Projekt, das Modelle aus der Klimamodellierung und den Erdsystemwissenschaften zusammenführt, um komplexe klimabezogene Fragestellungen zu erforschen. Über die Google Cloud werden mithilfe von Zarr öffentliche Datensätze bereitgestellt (Open Geospatial Consortium, 2022c, S. 8). Insgesamt zeigt sich, dass Zarr nicht explizit für geodatenspezifische Anforderungen entwickelt wurde. Trotzdem bietet es aufgrund seiner Flexibilität und Geschwindigkeit ein hohes Potenzial für die effiziente Verarbeitung großer Geo-Datensätze.

3.5 Analysemethoden raumzeitlicher Rasterdaten

Die Analyse raumzeitlicher Rasterdaten ist ein wesentlicher Bestandteil der modernen Geoinformationswissenschaften und Fernerkundung. Sie ermöglicht die Untersuchung dynamischer Prozesse und die Visualisierung von Veränderungen über Raum und Zeit. Sie bietet Einblicke in umweltbedingte Phänomene wie Vegetationsveränderungen, Landnutzungsänderungen und Urbanisierung. Die Verwendung raumzeitlicher Rasterdaten eröffnet zahlreiche Möglichkeiten, besonders durch die Integration und Analyse zeitbasierter Muster.

Auch die Analyse raumzeitlicher Rasterdaten steht durch die enormen Datenmengen vor erheblichen Herausforderungen. Die Vielfalt an Datenformaten, die Heterogenität und die gleichzeitige Verarbeitung räumlicher und zeitlicher Dimensionen erschwert die Modellwahl und Datenaufbereitung. Zudem erfordern anwendungsabhängige Algorithmen und die begrenzte Verfügbarkeit analysebereiter Datensätze flexiblere und skalierbare Analysemethoden (Sisodiya et al., 2023, S. 2009). Traditionelle Plattformen wie Google Earth Engine bieten zwar vorkonfigurierte Algorithmen und APIs für bestimmte Fernerkundungsanwendungen, lassen jedoch meist keine Integration benutzerdefinierter Algorithmen oder spezialisierter Bibliotheken wie Deep-Learning-Frameworks (z. B. PyTorch) zu (Xu, Du, Jian et al., 2022, S. 1). Dies zwingt oft zur Neuentwicklung spezialisierter Algorithmen, was die Effizienz und Anpassungsfähigkeit einschränkt. Zudem sind raumzeitliche Rasterdaten meist großflächig und enthalten umfangreiche Zeitreihen, die eine parallele Verarbeitung erfordern. Herkömmliche Systeme und das MapReduce-Paradigma stoßen hierbei an ihre Grenzen, da komplexe, mehrstufige Workflows und Datenabhängigkeiten durch spezifische Datenformate und Speicheranforderungen die parallele Verarbeitung erschweren (Xu, Du, Jian et al., 2022, S. 2). Die Analyse von raumzeitlichen Rasterdaten kann in datentrennbare und datenuntrennbare Berechnungen unterteilt werden. Datenuntrennbare Berechnungen wie etwa Klassifikationen erfordern Informationen aus dem gesamten Datensatz und können daher nicht einfach parallelisiert werden, da das Aufteilen der Daten zu Fehlern an Kachelkanten führt. Parallele Berechnungsverfahren für solche Analysen sind in der Regel individualisiert, was die Entwicklung einheitlicher paralleler Lösungen erschwert (Xu, Du, Fan et al., 2022, S. 1422). Eine weitere Herausforderung besteht in der Datenübertragung. Viele herkömmliche Plattformen und Algorithmen verarbeiten Daten ausschließlich im Arbeitsspeicher, was bei großen Projekten zu Speicherüberläufen und Skalierungsproblemen führen kann. Cloud-native Ansätze und parallele Verarbeitung bieten erste Lösungen für die Herausforderungen der Analyse großer raumzeitlicher Datenmengen. Häufig werden auch „on the fly“-Analysen eingesetzt, bei der Berechnungen und Transformationen direkt bei Bedarf durchgeführt werden, um die gleichzeitig zu verarbeitende Datenmenge moderat zu halten.

Die raumzeitliche Analyse lässt sich in drei Hauptkategorien einteilen (siehe Abbildung 16): deskriptive, prädiktive und präskriptive Analysen (Di und Yu, 2023, S. 165). In der Analyse von raumzeitlichen Rasterdaten spielen verschiedene Methoden eine zentrale Rolle. Zu den häufigsten Analysemethoden zählen Klassifikation, Clusteranalyse, Regression, Prädiktion, Anomalieerkennung, Musterverfolgung und Assoziationsregeln (Sisodiya et al., 2023, S. 1995). Die meisten dieser Algorithmen sind kachel- oder pixelbasiert (Xu, Du, Jian et al., 2022, S. 7f.). Die Klassifikation zielt darauf ab, jedem Pixel eine vordefinierte Klasse zuzuweisen, wobei entweder pixelbasierte spektrale Informationen oder objektbasierte Merkmale herangezogen werden. Dabei gibt es überwachte, unüberwachte und semi-überwachte Verfahren, die je nach Verfügbarkeit von Trainingsdaten eingesetzt werden. Das Clustering hingegen gruppiert Pixel auf Grundlage von Ähnlichkeiten und Unterschieden, ohne vordefinierte Klassen und umfasst verschiedene Ansätze wie partitionbasiertes, dichtebasiertes und hierarchisches Clustering.

Die deskriptive raumzeitliche Analyse beschäftigt sich mit beschreibenden Fragen und zielt darauf ab, verborgene Muster und Zusammenhänge in den Daten zu erkennen. Typische Analysemethoden umfassen Cluster- und Klassifikationsverfahren wie die unüberwachte K-Means-Clusterbildung zur Identifizierung ähnlicher Landschaftsstrukturen sowie spektral- oder objektbasierte Klassifikationen, etwa mithilfe von Random Forest oder Support Vector Machines (SVM), um Landnutzungsarten zu klassifizieren. Ein typisches Beispiel ist die Generierung von Labels für Deep-Learning-Modelle aus raumzeitlichen Ras-

terdaten zur Darstellung von Landbedeckungsänderungen über die Zeit (Nwaeze, 2024, o.S.). Solche Analysen werden in der Regel in Data Cubes ausgeführt. Dabei wird häufig von Data-Cube-Analysen gesprochen, die speziell auf die Analyse großer, raumzeitlicher Rasterdatenmengen bzw. Big Data ausgelegt sind und die effiziente Analyse umfangreicher Datensätze über Zeit und andere Dimensionen hinweg erlauben. Die Parallelverarbeitung dieser Data Cubes kann dann als eine Reihe unabhängiger Unteraufgaben betrachtet werden.

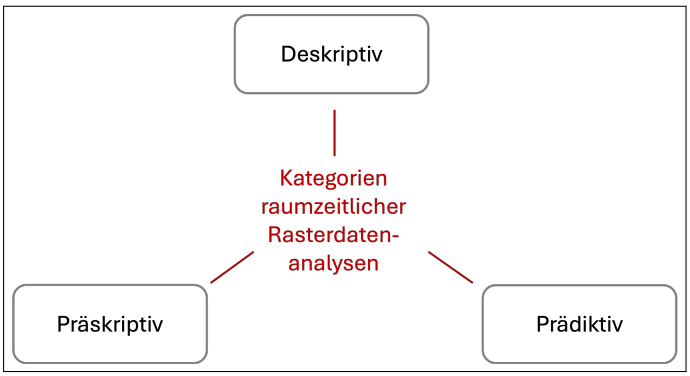

Abbildung 16: Kategorien raumzeitlicher Analysen (Eigene Darstellung nach Di und Yu, 2023, S.165)

Die prädiktive Analyse ist besonders auf Fernerkundungsdaten anwendbar, da sie die Möglichkeit bietet, die Phänomene im Laufe der Zeit kontinuierlich zu beobachten. Dabei werden zukünftige Entwicklungen durch Vorhersagemethoden ermittelt, die auf der Analyse und Extrapolation zeitlicher Muster beruhen. Ein zentraler Bestandteil dieser Analysen sind Zeitreihen, also Sammlungen sequentieller Datenpunkte, die zu festgelegten Zeitpunkten erhoben werden. Zeitreihen können regelmäßig (z. B. täglich oder monatlich) oder unregelmäßig gesammelt werden und ermöglichen es, Muster, Trends und Saisonalitäten in den Daten zu erkennen (Data Science Wizards, 2023, o.S.). Beispielsweise wurden Zeitreihen des NDVI und der Landoberflächentemperatur zur Vorhersage von Waldbränden verwendet. Ein anderes Beispiel ist die Verwendung eines Deep-Learning-Algorithmus, ein regionenbasiertes neuronales Faltungsnetzwerk (R-CNN) zur Vorhersage des Erdbeerertrags anhand von hochauflösenden Luftbildern (Di und Yu, 2023, S. 158). Die Anwendung dieser Methoden auf große raumzeitliche Rastderdatenmengen birgt jedoch erhebliche Herausforderungen, darunter hohe Rechenanforderungen, komplexe Trainingsprozesse und das Risiko von Overfitting. Zudem benötigen sie oft einen vollständigen Zugriff auf den gesamten Datensatz und erfordern iterative Prozesse, was bei sehr großen Datenmengen ineffizient und ressourcenintensiv ist.

Präskriptive Analysen befassen sich damit, welche Maßnahmen ergriffen werden sollten, um die erwarteten Ergebnisse zu erzielen. Diese Analysen nutzen Simulationsmodelle und Entscheidungsalgorithmen, um die Effizienz von Maßnahmen zu bewerten und fundierte Entscheidungen zu treffen. Hier kommen zunehmend maschinelle Lernverfahren wie Deep Learning und Optimierungsalgorithmen zum Einsatz, um Entscheidungen in komplexen Umgebungen wie der Planung von Klimaanpassungsmaßnahmen oder der Landwirtschaftsoptimierung zu unterstützen (Di und Yu, 2023, S. 156f.).

Diese Analysen können anschließend mithilfe geeigneter Visualisierungstechniken dargestellt werden, um die Ergebnisse anschaulich zu präsentieren und fundierte Entscheidungen zu unterstützen. Wichtige Faktoren bei der Visualisierung sind dabei die Wahl der Dimension (2D, 3D oder 4D) und die Art der Darstellung (statisch oder dynamisch). Die raumzeitliche Analyse von Rasterdaten bietet ein enormes Potenzial für eine Vielzahl von Anwendungsbereichen, von der Umweltüberwachung bis hin zur Stadtplanung und Landwirtschaft. Obwohl bereits Fortschritte erzielt wurden, befinden sich viele der zugrunde liegenden Technologien und Methoden noch in der Entwicklung. Die kontinuierliche Verbesserung von Algorithmen, Rechenleistung und Datenverfügbarkeit wird in Zukunft dazu beitragen, die Effizienz und Nutzung raumzeitlicher Analysen weiter zu steigern und damit noch tiefere Einblicke in dynamische Prozesse und Veränderungen zu ermöglichen.

4 Methodik

Aufbauend auf den theoretischen Grundlagen und Konzepten aus Kapitel 3 beschreibt dieses Kapitel die Methodik zur Evaluierung der cloud-optimierten Datenformate COG und Zarr hinsichtlich ihrer Eignung für die Speicherung und Analyse großer raumzeitlicher Rasterdaten. Im Mittelpunkt steht dabei die Entwicklung eines Konzepts sowie der Aufbau einer geeigneten Architektur, die als zentrale Grundlage für den Vergleich dienen. Zunächst wird die Datengrundlage definiert, um die Formate mit praxisrelevanten raumzeitlichen Rasterdaten untersuchen zu können. Eine Anforderungsanalyse identifiziert die zentralen Kriterien für die Bewertung der Formate. Dabei werden aus den zuvor erarbeiteten Grundlagen Kriterien abgeleitet, die für die effiziente Speicherung und den Zugang zu raumzeitlichen Rasterdaten entscheidend sind. Das Untersuchungskonzept beschreibt das methodische Vorgehen, einschließlich der Entwicklung verschiedener Testszenarien mit variierenden Dateieigenschaften. Ziel ist es, sowohl die Speicherung in der Cloud als auch den Zugriff auf die Daten und deren Analyse umfassend zu evaluieren. Ein weiterer Schwerpunkt dieses Kapitels liegt auf der Entwicklung einer passenden Infrastruktur, die die Evaluierung technisch ermöglicht. Ziel ist es, die Stärken und Schwächen der Formate in einem praxisnahen Umfeld herauszuarbeiten, das Cloud Computing, parallele und verteilte Verarbeitung vereint, um Cloud-native Geospatial-Paradigmen umzusetzen. Durch die detaillierte Beschreibung der verwendeten Technologien und der gesamten Architektur wird sichergestellt, dass die spezifischen Eigenschaften der Formate unter realistischen Bedingungen getestet werden können. Dieses Kapitel schafft somit die methodischen und technischen Grundlagen für die theoretische Vergleichsanalyse (Kapitel 5) sowie die praktische Implementierung (Kapitel 6) und stellt sicher, dass die Untersuchung fundiert und praxisnah durchgeführt werden kann.

4.1 Datengrundlage

Das LGLN stellt seit über 60 Jahren flächendeckende Luftbildaufnahmen von Niedersachsen bereit (Landesamt für Geoinformation und Landesvermessung Niedersachsen, n. d., o.S.). Die Luftbilder werden mit hochauflösenden Kameras aus Flugzeugen aufgenommen, wobei Niedersachsen in fortlaufenden, überlappenden Streifen beflogen wird, um eine lückenlose Erfassung des Geländes zu gewährleisten. Der aktuelle Befliegungsturnus beträgt drei Jahre. Seit 2008 werden aus diesen orientierten Luftbildern DOPs generiert, die als verzerrungsfreie und maßstabsgetreue Rasterdaten der Erdoberfläche fungieren. Die Generierung der DOPs erfolgt durch ein rechnergestütztes Verfahren, bei dem die Luftbilder orientiert und auf das digitale Geländemodell projiziert werden. Diese hochauflösenden Abbildungen werden nach einem bundeseinheitlichen Produktstandard aufbereitet und jährlich für ein Drittel der Landesfläche Niedersachsens bereitgestellt. Seit 2021 hat das LGLN die traditionellen DOPs durch TrueDOPs ersetzt, die mit einem bildbasierten digitalen Oberflächenmodell erstellt werden (Landesamt für Geoinformation und Landesvermessung Niedersachsen, 2021, o.S.). Diese TrueDOPs minimieren sichttote Bereiche, Umklappeffekte und Verdeckungen und ermöglichen eine grundrisstreue Darstellung der Objekte. Dadurch wird eine höhere Interoperabilität mit anderen Geobasisdaten erreicht. DOPs stellen mittlerweile eine zentrale Datenquelle für zahlreiche Anwendungsbereiche dar, insbesondere in der Landwirtschaft, im Umweltschutz, in der Stadtplanung sowie in der wissenschaftlichen Forschung.

Seit ihrer Einführung als Open-Data 2021 sind die DOPs öffentlich in Form von cloud-optimierten Geotiff und JPEG-Dateien zugänglich, was zu einem Anstieg der Nutzung in verschiedenen Disziplinen geführt hat (Landesamt für Geoinformation und Landesvermessung Niedersachsen, 2021, S. 3). Dies erfolgt gemäß der Open-Data-Strategie und den Vorgaben der Arbeitsgemeinschaft der Vermessungsverwaltungen der Bundesrepublik Deutschland (AdV), sodass die DOPs unkomprimiert und kostenfrei zur Verfügung stehen. Die DOPs des LGLN sind somit für die Evaluation cloud-optimierter Rasterdatenformate geeignet. Der große Datensatz, der spektrale Informationen über das gesamte Landesgebiet und über verschiedene Zeitpunkte hinweg bereitstellt, ermöglicht eine fundierte Bewertung der Nutzbarkeit, des Zugriffs und der Effizienz dieser Datenformate im Rahmen von raumzeitlichen Anwendungsfällen. Dadurch wird die Nutzung praxisorientierter Daten in der folgenden Untersuchung ermöglicht und bietet darüber hinaus eine wertvolle Gelegenheit, den Zugriff und die Nutzbarkeit der Open-Data-

Bereitstellung mit der neu eingeführten STAC-API zu testen. Dies ermöglicht eine Bewertung ihrer Eignung für raumzeitliche Anwendungen und gibt gleichzeitig Rückmeldung darüber, wie gut die Daten in dieser Form für praktische Analysen genutzt werden können.

Die DOPs bieten durch ihre Standardisierung und hohe Datenqualität eine solide Grundlage für Analysen, wie die Erkennung von Mustern, die Ableitung von Vorhersagen oder die Untersuchung von Vegetationsveränderungen. Sie sind einheitlich georeferenziert (ETRS89/UTM32), geometrisch und radiometrisch korrigiert sowie in 2 × 2 km großen Kacheln mit 20 cm Bodenauflösung verfügbar (Landesamt für Geoinformation und Landesvermessung Niedersachsen, 2021, S. 2f.). Überlappungen zwischen den Luftbildern wurden bei der Erstellung der DOPs eliminiert. Es stehen 3-Kanal-Echtfarbbilder (RGB) und 4-Kanal-Multispektralbilder (RGBI) mit einem Wertebereich von 8 Bit pro Farbkanal bereit. Die geometrische Genauigkeit beträgt ± 0,4 m und die Metadaten sind gemäß den AdV-Standards strukturiert. Aufgrund ihrer Vorbereitung können DOPs deshalb im weiteren Sinne als ARD betrachtet werden, da sie eine standardisierte und vorverarbeitete Datenbasis bieten, die eine sofortige Nutzung ermöglicht. Sie folgen jedoch nicht exakt dem Earth-Observation-Paradigma und spezifischen satellitengestützten Vorgaben, sondern beruhen auf anderen, regionalen Standards der AdV, was bei der Untersuchung berücksichtigt werden muss. Zudem hat die Umstellung auf TrueDOP zu Qualitätsunterschieden geführt. Trotz ihrer hohen Auflösung und Standardisierung erfordern die DOPs bei raumzeitlichen Analysen eine Berücksichtigung herstellungsbedingter Faktoren. Da jährlich nur ein Drittel der Landesfläche erfasst wird, entstehen unvollständige jährliche Datensätze, es gibt überlappende Befliegungsgrenzen und kleinräumige Unterschiede in den Aufnahmezeitpunkten. Diese führen zu unregelmäßigen Zeitreihen, die bei der Erstellung von Data Cubes berücksichtigt werden müssen, bieten jedoch gleichzeitig wertvolle Erkenntnisse zur praktischen Nutzbarkeit und Optimierungsmöglichkeiten der Daten.

4.2 Anforderungsanalyse

Um die Zugänglichkeit zu der wachsenden Mengen an raumzeitlichen Rasterdaten zu verbessern, sind cloud-optimierte Formate ein wichtiger Schritt. Sie bieten effiziente Lösungen für den Umgang mit großen Geodatensätzen, ermöglichen eine direkte Verarbeitung der Daten aus dem Objektspeicher ohne lokale Speicherung und liefern Vorteile wie hohe Verfügbarkeit, skalierbare Leistung und geringe Latenz. Dennoch bringen raumzeitliche Rasterdaten durch die großen Datenmengen und die rechnintensiven Operationen einige Herausforderungen für cloud-optimierte Rasterdatenformate mit sich. Die Anforderungen an solche Formate sind vielfältig und betreffen sowohl die effiziente Speicherung als auch die nahtlose Integration in Cloud-Workflows durch einen performanten Zugriff auf die Daten. In diesem Kapitel werden die zentralen Anforderungen beschrieben, die an die cloud-optimierten Rasterdatenformate COG und Zarr gestellt werden, um deren Eignung für die Speicherung und Analyse raumzeitlicher Rasterdaten zu evaluieren. Hier werden, aufbauend auf den theoretischen Grundlagen und den Grundlagen der Dateiformate, technische und funktionale Anforderungen erarbeitet, die das Fundament für die nachfolgenden Untersuchungen bilden.

Anforderung	**Technische Umsetzung**	**Funktionale Bedeutung**
1. Partitionierung und partieller Zugriff	Das Format muss Daten in Partitionen als Chunks oder Tiles speichern können. Erforderlich sind Strukturen zur Definition von Position und Größe jeder Partition sowie Metadaten zur Verwaltung der Partitions-Beziehungen. Dafür sind effiziente Indexierungs- und Verweismechanismen bereitzustellen.	Ermöglicht den gezielten Zugriff auf räumliche und zeitliche Teilbereiche großer Rasterdaten, wodurch Ladezeiten minimiert und spezifische Regionen und Zeitintervalle ausgewählt werden können. Unterstützt parallele und verteilte Verarbeitung für raumzeitliche Analysen.

2. Speicherreduktion	Das Format muss verschiedene verlustfreie und verlustbehaftete Kompressionsalgorithmen unterstützen, die sich an die Datencharakteristika anpassen lassen. Zudem sollte eine effiziente Nullwertbehandlung vorgenommen werden können, um Speicherplatz zu sparen.	Dies reduziert den Speicherbedarf und Übertragungskosten, ohne dabei die Qualität der Daten zu verringern. Dies erleichtert die Datenverarbeitung bei großen raumzeitlichen Datensätzen, insbesondere bei heterogenen Daten mit vielen Nullwerten.
3. Skalierbarkeit	Das Format muss flexibel und effizient mit unterschiedlichen, insbesondere sehr großen, Datenmengen umgehen können. Es sollte die Unterstützung mehrdimensionaler Strukturen ermöglichen und die Erweiterung um neue Dimensionen oder Daten erlauben, ohne die Datenstruktur neu organisieren zu müssen.	Ermöglicht die Handhabung wachsender Datenmengen und das Hinzufügen neuer Zeitpunkte, räumlicher Ausschnitte, weiteren Dimensionen oder Attribute, ohne die Integrität oder Performance zu beeinträchtigen.
4. Modifizierbarkeit	Das Format muss Datenänderungen und -ergänzungen effizient unterstützen. Technisch sind Mechanismen für parallele Lese- und Schreibvorgänge sowie ggf. Transaktionen und Sperren erforderlich, um die Datenintegrität in verteilten Systemen zu gewährleisten.	Ermöglicht dynamische Anpassungen an raumzeitlichen Rasterdaten, wie das Hinzufügen neuer Zeitpunkte oder Änderungen räumlicher Ausschnitte, ohne einen vollständigen Neuschreibaufwand. Dies fördert eine flexible „on the fly“-Verarbeitung und ermöglicht die Harmonisierung großer Datensätze.
5. Raumzeitliche Datenstrukturen	Das Format muss raumzeitliche Rasterdaten unterstützen. Es muss sowohl eine räumliche Dimension, eine zeitliche Dimension in verschiedenen Auflösungen als auch nicht-räumliche Attribute und Metadaten speichern können. Dafür sind mehrdimensionale Strukturen und effiziente Indexierung notwendig.	Ermöglicht die Speicherung und Abfrage raumzeitlicher Rasterdaten, z. B. zur Analyse von Veränderungen über Raum und Zeit sowie zur Erkennung von Trends. Das Format muss Abfragen in verschiedenen Kombinationen und Reihenfolgen der Dimensionen unterstützen (zeitlich, räumlich, raumzeitlich, attributiv, metadatenbasiert).
6. Standardisierung und Interoperabilität	Das Format muss standardisiert sein und verschiedene Koordinatenreferenzsysteme, Datenmodelle sowie Schnittstellen unterstützen. Es muss kompatibel mit Metadatenstandards, OGC-Standards, Geoinformationssystem (GIS)-Plattformen (z. B. QGIS, ArcGIS) und Verarbeitungsbibliotheken wie GDAL sein.	Dies fördert die Interoperabilität und ermöglicht die einfache Nutzung in GIS-Systemen und anderen Analysetools sowie den Datenaustausch zwischen verschiedenen Plattformen unter Einhaltung geographischer Standards.

7. Übersichten	Das Format sollte Bildpyramiden und Übersichten auf unterschiedlichen Zoomstufen speichern. Neben räumlichen Resampling sollten auch zeitlichen Aggregationen möglich sein (z. B. Tages-, Monats-, Jahresübersichten)	Optimiert für räumliches und zeitliches Resampling zur Analyse unterschiedlicher Detailstufen. Zudem wird eine schnelle Visualisierung und die Nutzung in GIS-Analysen mit verschiedenen Zoomstufen ermöglicht.

Tabelle 2: Zusammenfassung der Anforderungen an cloud-optimierte Rasterdatenformate

Anhand der in Tabelle 2 beschriebenen Anforderungen lassen sich vier übergeordnete Bewertungskriterien ableiten, die sowohl die theoretische als auch die praktische Eignung der untersuchten cloud-optimierten Datenformate bewerten:

- **Benutzerfreundlichkeit:** Dieses Kriterium bezieht sich auf die Einfachheit und Intuitivität der Nutzung des Datenformats. Aspekte wie leichte Modifizierbarkeit (Anforderung 2), Kompatibilität mit gängigen Tools und Standards (Anforderung 2) sowie Unterstützung raumzeitlicher Datenstrukturen (Anforderung 2) und Übersichten (Anforderung 2) spielen dabei eine Rolle, genauso wie die praktische Handhabung der Formate im Hinblick auf den Implementierungsaufwand.
- **Effizienz:** Dieses Kriterium bezieht sich auf die optimale Nutzung von Ressourcen wie Speicherplatz, Rechenzeit und Netzwerkbandbreite. Dabei spielen die Speicherreduktion (Anforderung 2) und die Performance bei partiellen Zugriffen (Anforderung 2) sowie die Fähigkeit zur effizienten Parallelverarbeitung eine entscheidende Rolle. Die effiziente Kompression und der gezielte Zugriff auf benötigte Datenbereiche tragen wesentlich zur Effizienz bei.
- **Skalierbarkeit:** Dieses Kriterium bewertet die Fähigkeit des Datenformats, mit wachsenden Datenmengen (Anforderung 2) und komplexeren Datenstrukturen umzugehen, ohne dass die Performance oder Datenintegrität leidet. Auch die Flexibilität der Formate ist dabei entscheidend.
- **Robustheit:** Unter Robustheit wird die Stabilität der Performance, die Konsistenz der Ergebnisse sowie die Datenintegrität verstanden. Insbesondere wird bewertet, wie zuverlässig die Formate unter verschiedenen Bedingungen arbeiten (Anforderung 2) und ob Mechanismen zur Sicherstellung der Datenintegrität vorhanden sind, um Fehler bei parallelen Lese- und Schreibvorgängen (Anforderung 2) zu vermeiden.

Diese Kriterien ermöglichen eine strukturierte Bewertung der Formate hinsichtlich ihrer funktionalen und technischen Eignung und stellen sicher, dass sowohl theoretische Anforderungen als auch praktische Erkenntnisse berücksichtigt werden.

4.3 Untersuchungskonzept

Die vorliegende Untersuchung verfolgt das Ziel, die Leistungsfähigkeit der cloud-optimierten Rasterdatenformate Zarr und COG im Kontext raumzeitlicher Rasterdaten zu evaluieren. Der Schwerpunkt liegt auf der Untersuchung der Handhabung und der spezifischen Eigenschaften dieser Formate, insbesondere hinsichtlich der zeitlichen Dimension. Zur praktischen Evaluierung der cloud-optimierten Datenformate wird ein eigenständiges Anwendungsszenario implementiert. Das Szenario umfasst die Aufbereitung und Speicherung der DOP im Zarr- und COG-Format in einem Cloud-Objektspeicher. Anschließend wird auf die Dateien im Cloud-Objektspeicher zugegriffen, um raumzeitliche Abfragen und Analysen durchzuführen.

Theoretischer Vergleich
Um eine fundierte Grundlage für die praktische Evaluierung zu schaffen, wird zunächst ein theoretischer Vergleich anhand der verfügbaren Dokumentationen vorgenommen. Dieser dient dazu, die Architektur, die Speicherstrukturen und die Zugriffsmechanismen der Formate zu untersuchen und zentrale

Gemeinsamkeiten und Unterschiede herauszuarbeiten. Der Vergleich wird anhand der in Kapitel 4.2 aufgeführten Anforderungen vorgenommen. Dies umfasst neben technischen Kriterien wie dem partiellen Zugriff, Modifizierbarkeit und der Kompression auch die Einschätzung der Interoperabilität sowie die Unterstützung der Zeitdimension und raumzeitlichen Abfragemechanismen. Durch die Identifizierung dieser Aspekte lassen sich erste Rückschlüsse auf die Eignung der Formate für die Speicherung und Analyse raumzeitlicher Rasterdaten ziehen. Zudem liefern die theoretischen Erkenntnisse eine wesentliche Grundlage für die nachfolgende praktische Evaluierung und zeigen Grenzen und Hindernisse auf.

Praktische Evaluierung
Die praktische Evaluierung basiert auf den in Kapitel 4.1 vorgestellten DOPs des LGLN, die durch ihre regelmäßige Erhebung, hohe räumliche und zeitliche Auflösung sowie Standardisierung eine geeignete Datengrundlage bieten. Dabei wird auch die Nutzbarkeit und der Zugriff auf die kostenfreien cloud-optimierten Daten des LGLN praxisnah untersucht. In diesem Abschnitt wird praktisch untersucht, wie die Formate COG und Zarr die Anforderungen aus Kapitel 4.2 erfüllen, mit Fokus auf partiellen Zugriff, Kompression und Unterstützung raumzeitlicher Abfragen. Dabei wird das COG-Format für die Speicherung und Analyse der ARD eingesetzt, während das Zarr-Format für die Speicherung von Data Cubes aus den DOPs verwendet wird. Die Evaluation fokussiert primär auf die Konzepte und die praktische Handhabung der Formate und zielt weniger umfangreiche Big-Data-Analysen ab. Deshalb wird der Umfang der Datensätze moderat gehalten. Dies entspricht den Dask-Best-Practices und erleichtert die erste Implementierung sowie Analyse, ohne dass eine übermäßige Belastung der Rechenressourcen entsteht. Die Datenmengen sind jedoch so gewählt, dass sie die Kapazität eines lokalen Arbeitsspeichers überschreiten und eine parallele und verteilte Verarbeitung erfordern. Das Untersuchungsgebiet liegt im niedersächsischen Teil des Harzes, einer Region mit hoher Biodiversität und dynamischen Veränderungen durch Borkenkäferbefall und Waldbrände. Diese vielfältigen Einflüsse machen den Harz zu einem idealen Studiengebiet für raumzeitliche Analysen. Zur Evaluation von Performance und Skalierbarkeit gemäß Anforderung A3 aus Kapitel 4.2 werden drei Datenmengen zwischen 4 und 40 GB (10 bis 100 DOPs) getestet, wobei für beide Formate identische räumliche und zeitliche Abdeckungen gewählt werden. Für konsistente Zeitreihen werden Nullwerte und Überlappungsbereiche der Befliegungszonen ausgeschlossen. Unter Berücksichtigung dieser Anforderungen ergeben sich die in Tabelle 3 dargestellten Datenmengen.

Kategorie	**Anzahl der DOPs**	**Anzahl der Zeitpunkte**	**Räumliche Ausdehnung (Pixel)**	**Geschätzte Datenmenge (GB)**
kleine Datenmenge	9	3	30.000 x 10.000	4
mittlere Datenmenge	54	3	60.000 x 30.000	22
große Datenmenge	100	4	50.000 x 50.000	40

Tabelle 3: Einteilung der Testdaten

4.3.1 Speicherung in der Cloud

Die Speicherung der DOPs in cloud-optimierten Formaten erfolgt, wie in Abbildung 17 dargestellt, in drei Schritten: Laden der DOP mit STAC, Vorbereitung und Upload der Daten im COG- als auch im Zarr-Format in einem Cloud-Bucket. Im Rahmen dieses Prozesses werden die beiden Formate anhand der Anforderungen aus Kapitel 4.2 geprüft. Dabei wird bei der Datenaufbereitung darauf geachtet, dass wichtige strukturelle Anforderungen erfüllt werden, insbesondere die Unterstützung von räumlichen und zeitlichen Dimensionen sowie die Verwaltung von Attributen und Metadaten (vgl. Anforderung 2). Der Fokus liegt dabei darauf, wie jedes Format diese Anforderungen unterstützt und wie die Strukturierung der Daten erfolgen muss, um diese Informationen korrekt zu speichern. Außerdem wird die Generierung von Übersichten aus Anforderung 2 dabei evaluiert. Die Durchführung erfolgt mit den in Abschnitt 4.4 beschriebenen Technologien. Zusätzlich wird die Partitionierung und Kompression genauer evaluiert, da diese Parameter direkten Einfluss auf die Uploadgeschwindigkeit und die resultierende Dateigröße haben (vgl. Anforderung 2 und 2). Um den Einfluss dieser beiden Faktoren besser zu verstehen, werden separate

Testszenarien für die Partitionierung und die Kompression durchgeführt. Dabei kommen unterschiedliche Einstellungen zum Einsatz, um die optimale Balance zwischen Speicherbedarf und Upload-Performance zu identifizieren.

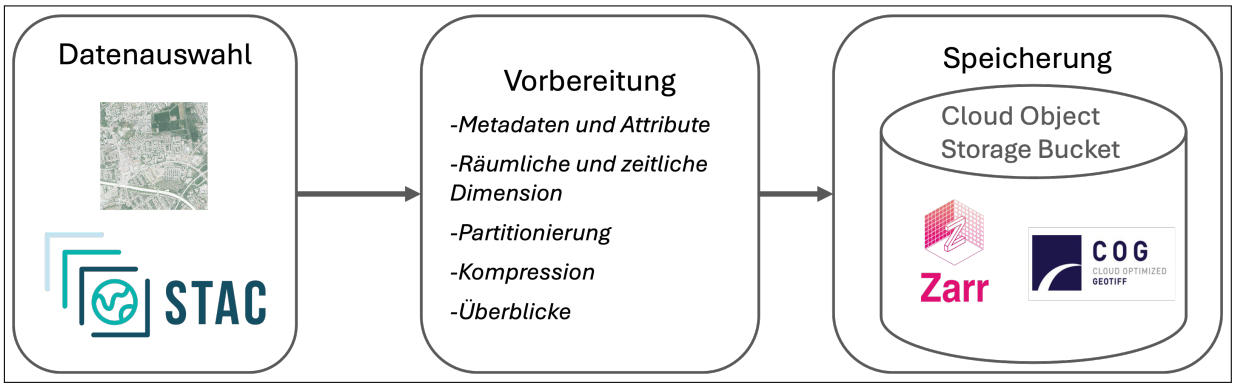

Abbildung 17: Untersuchungsablauf zum Upload cloud-optimierter Rasterdateien (Eigene Darstellung)

Partitionierung
Die Partitionierung ist zentral für die Optimierung der Speicherung und des Zugriffs auf große raumzeitliche Rasterdaten in cloud-optimierten Formaten. Sie ermöglicht sowohl den gezielten Zugriff auf Teilmengen als auch die parallele Verarbeitung großer Datenmengen. Kleinere Partitionsgrößen ermöglichen oftmals einen schnelleren Zugriff auf Teilbereiche, während größere Partitionsgrößen bei der Speicherung effizienter sind und den Speicherbedarf reduzieren können. Ziel dieser Analyse ist es, den Einfluss unterschiedlicher räumlicher Partitionsgrößen auf Dateigröße, Uploadgeschwindigkeit, CPU-Auslastung und Durchsatz zu untersuchen, um eine effiziente Balance zwischen schneller Datenübertragung und optimaler Speichernutzung zu identifizieren. Dabei wird untersucht, wie die Datenmenge und -struktur die Wahl der Partitionsgröße beeinflussen. In dieser Arbeit wird ausschließlich die räumliche Partitionierung untersucht, da das COG-Format eine räumliche Partitionierung vorgibt und die geringe Anzahl verfügbarer Zeitpunkte in den verwendeten Daten eine zeitliche Partitionierung kaum zulässt.

Für die Analyse werden verschiedene Partitionsparameter definiert und auf die Erzeugung von COG- und Zarr-Dateien angewandt, basierend auf den in Kapitel 4.3 beschriebenen Datenmengen. Die Partitionsgrößen berücksichtigen sowohl die Anforderungen der Formate als auch die Eigenschaften der Ausgangsdaten. Bei einer DOP-Auflösung von 10.000 x 10.000 Pixeln pro Band wird eine maximale Kachelgröße von 10.000 Pixeln als Obergrenze gewählt. Entsprechend der Spezifikationen des COG-Formats muss die Kachelgröße durch 16 teilbar sein. Die untersuchten Partitionsgrößen orientieren sich daher an Zweierpotenzen. Da die Standardgröße für COG-Kacheln 512 x 512 Pixel beträgt und die Overviews üblicherweise bis auf 256 x 256 Pixel reduziert werden, stellen dies die kleinsten untersuchten Partitionsgrößen dar. Zusätzlich werden Partitionsgrößen gewählt, die sowohl durch 16 teilbar als auch Vielfache der maximalen Pixelanzahl von 10.000 Pixeln sind, um den Einfluss der Datenstruktur auf die Partitionierung zu bewerten. Die Verwendung identischer Partitionsgrößen für beide Formate gewährleistet eine direkte Vergleichbarkeit der Ergebnisse und ermöglicht eine objektive Bewertung der spezifischen Stärken jedes Formats im Hinblick auf Speicher- und Zugriffsperformance. Daraus ergeben sich die folgenden räumlichen Partitionsgrößen, die beim Upload von COG- und Zarr-Dateien verwendet werden: 256x256, 400x400, 512x512, 1.024x1.024, 2.000x2.000, 4.096x4.096, 8.192x8.192 und 10.000x10.000.

Kompression
Die Kompression spielt eine zentrale Rolle bei der Speicherung großer raumzeitlicher Rasterdaten, da sie eine effizientere Datenübertragung und eine Reduzierung der Dateigröße ermöglicht, ohne die wesentlichen Informationen zu verlieren. Grundsätzlich wird zwischen verlustfreier und verlustbehafteter Kompression unterschieden. Verlustfreie Kompression erlaubt die exakte Wiederherstellung der Originaldaten und ist für wissenschaftliche Anwendungen essenziell, bei denen die numerische Integrität der Daten entscheidend ist. Im Gegensatz dazu reduziert verlustbehaftete Kompression die Datenmenge durch Nä-

herungen oder das Entfernen nicht wesentlicher Informationen, was vor allem für Visualisierungs- oder Generalisierungszwecke genutzt wird.

Da präzise Werte in der Rasterdatenanalyse unabdingbar sind, wird in dieser Untersuchung ausschließlich verlustfreie Kompression eingesetzt. Die ausgewählten Algorithmen werden hinsichtlich ihres Einflusses auf die Dateigröße, die Uploadgeschwindigkeit und das Kompressionsverhältnis getestet. Das Kompressionsverhältnis, definiert als Verhältnis der ursprünglichen zur komprimierten Dateigröße, dient als Maß für die Effizienz eines Verfahrens (z. B. 10:1 für eine Reduktion von 20 Megabyte (MB) auf 2 MB). Die Tests werden mit den drei unterschiedlichen Datenmengen aus Kapitel 4.3 durchgeführt. Ziel der Analyse ist es, Algorithmen zu identifizieren, die eine optimale Balance zwischen Speicherplatzreduktion und Verarbeitungszeit bieten, ohne die Datenintegrität zu gefährden. Um den Einfluss der Partitionierung auf die Kompression zu bewerten, werden die Tests für kleine (512 x 512 Pixel) und große (10.000 x 10.000 Pixel) Partitionsgrößen durchgeführt. Dabei kommen ausschließlich verlustfreie Kompressionsalgorithmen zum Einsatz, die für das COG- und Zarr-Format geeignet sind. Die Auswahl der Kompressionsalgorithmen beruht auf typischen Empfehlungen aus der Literatur, insbesondere dem Cloud-Optimized Geospatial Formats Guide (Cloud-Native Geospatial Foundation, 2023, o.S.) sowie Arbeiten von Alberti, 2018, o.S. Für COG werden LZW, Deflate und ZSTD ausgewählt, die laut Alberti, 2018 besonders effektiv für COG sind. Die Algorithmen unterstützen zudem die Verwendung von Prädiktoren, bei denen nur die Differenz zum vorherigen Wert anstelle des tatsächlichen Wertes gespeichert wird. Dabei werden drei Prädiktoreinstellungen unterschieden: kein Prädiktor, horizontale Differenzierung und Gleitkomma-Vorhersage. Die Verwendung von Prädiktoren bietet den Vorteil, dass er wiederholte Muster entlang von Bildachsen besser komprimieren kann, was besonders bei regelmäßigen Strukturen in den DOPs von Vorteil ist. Da die untersuchten Daten ausschließlich ganzzahlige Werte aufweisen, werden in dieser Untersuchung nur die Prädiktoren 1 und 2 getestet. Für Zarr werden LZ4, ZSTD und Blosc verwendet, die in Cloud-Umgebungen als effizient für große multidimensionale Daten gelten (Cloud-Native Geospatial Foundation, 2023, o.S.). Im Folgenden eine kurze Beschreibung der gewählten Algorithmen und ihrer Funktionsweise:

- **Deflate:** Der Deflate-Algorithmus wird in zahlreichen Geodatenformaten, darunter GeoTIFF, standardmäßig eingesetzt. Er basiert auf einer Kombination der LZ77-Kompression und der Huffman-Kodierung, was ihm eine ausgewogene Leistung in Bezug auf Komprimierungsrate und Geschwindigkeit verleiht (Daniel O'Donohue, 2023, o.S.).

- **LZW (Lempel-Ziv-Welch):** LZW ist ein verlustfreier Algorithmus, der durch die Erkennung und Kodierung wiederkehrender Muster effizient komprimiert. LZW basiert auf einer Wörterbuch-basierten Kompression (Daniel O'Donohue, 2023, o.S.). Diese Methode eignet sich gut für strukturierte und homogene Rasterdaten.

- **ZSTD (Zstandard):** ZSTD ist ein moderner Algorithmus, der speziell für Anwendungen in Cloud-Umgebungen und datenintensive Prozesse entwickelt wurde. Er bietet eine hohe Kompressionsrate bei schneller Dekompression (Daniel O'Donohue, 2023, o.S.).

- **LZ4:** LZ4 ist bekannt für seine extrem hohe Geschwindigkeit bei der Kompression und Dekompression und wird häufig in Szenarien eingesetzt, in denen schnelle Datenzugriffe wichtiger sind als maximale Kompression. LZ4 verwendet wie viele andere Kompressionsalgorithmen die LZ77-Methode, um wiederkehrende Zeichenfolgen zu ersetzen, jedoch ohne zusätzliches Huffman- Encoding (LZ4 developers, 2024, o.S.).

- **Blosc:** Blosc ist ein Meta-Kompressionsalgorithmus, der Techniken wie Byte-Shuffling integriert und andere Algorithmen wie LZ4 oder ZSTD einbindet, um die Kompressionseffizienz weiter zu steigern. Blosc wurde speziell für die schnelle Verarbeitung großer binärer Daten entwickelt und ist daher ideal für das Zarr-Format. Blosc reduziert nicht nur die Größe großer Datensätze, sondern beschleunigt durch die Blockierungstechnik dabei auch speichergebundene Berechnungen (Blosc Developers, 2024, o.S.).

4.3.2 Analyse raumzeitlicher Rasterdaten

Im letzten Teil des Untersuchungskonzepts steht die Analyse großer raumzeitlicher Rasterdaten im Fokus. Das Ziel ist es, die Effizienz der Formate für raumzeitliche Anwendungen in einer Cloud-Umgebung zu bewerten. Diese Analyse wird in zwei Teile unterteilt:

Teil 1: Raumzeitliche Abfragen
Im ersten Teil erfolgt die Untersuchung der Zugriffs- und Lade-Performance. Der Fokus liegt dabei darauf, wie sich die verschiedenen Parameter der Partitionierung und Kompression, die in Kapitel 4.3.1 beschrieben werden, auf die Zugriffsgeschwindigkeit auswirken. Darüber hinaus wird die Datenstruktur von ARD und Data Cubes im Bezug auf raumzeitliche Analysen evaluiert. Diese Tests berücksichtigen Anforderungen aus dem Kapitel 4.2 wie die Möglichkeit des partiellen Zugriffs (Anforderung 2) und die Effizienz der Speicherreduktion (Anforderung 2) sowie die Skalierbarkeit (Anforderung 2) durch die Verwendung verschiedener Datenmengen. Ein weiterer Aspekt dieser Untersuchung ist die Analyse, inwieweit die Reihenfolge und Kombinationen von Dimensionen die Abfragegeschwindigkeit beeinflussen. Verschiedene Abfragen über die räumliche und zeitliche Dimension werden getestet, um zu prüfen, ob das Format eine Flexibilität für raumzeitliche Abfragen bietet und ob bestimmte Dimensionen oder deren Reihenfolge den Zugriff verlangsamen oder beschleunigen. Zur Bewertung der Effizienz und Performance der Analysen werden die Zugriffsgeschwindigkeit, die Anzahl der Tasks, die Anzahl der HTTP-Requests und die übertragende Datenmenge betrachtet.

Zuerst wird untersucht, wie unterschiedliche Kachelgrößen und Kompressionsmethoden die Zugriffs- und Lade-Performance beeinflussen. Ziel ist es, die optimale Konfiguration für eine effiziente Verarbeitung von DOPs zu identifizieren. Hierbei wird sowohl die Größe des Data Cubes als auch die Größe des Abfragebereichs berücksichtigt (siehe Tabelle Tabelle 4). Für Zarr wird dabei ein kleiner (9 DOPs) und ein großer Data Cube (100 DOPs) verwendet. Im Fall von COGs liegen die Bilddaten einzeln vor. Üblicherweise ist bekannt, welche DOPs benötigt werden oder die Daten werden mit entsprechenden Suchinstanzen, wie beispielsweise STAC, gefiltert. Deshalb wird angenommen, dass die relevanten Dateien bekannt sind, weshalb eine Unterteilung der Datenmenge entfällt. Die Tabellenzeile Datenmenge ist entsprechend nur für Zarr-Dateien relevant. Für die Untersuchung werden die in Tabelle 4 aufgelisteten Kachelgrößen und Kompressionsmethoden verwendet. Die Kompressionsmethoden von COG stehen dabei in Klammern. Der Zusatz P2 meint dabei die Verwendung des Prädiktors 2. Die Abfragebereiche sind so gewählt, dass sowohl kleine als auch große räumliche und zeitliche Ausdehnungen berücksichtigt werden. Der kleine räumliche Bereich umfasst dabei ca. die Größe eines DOPs, während der große Bereich ca. 16 DOPs umfasst.

	Chunking	**Kompression**	**Zarr / COG**					
Datenmenge (Anzahl DOPs)			**9**		**100**			
Testnummer			**1**	**2**	**3**	**4**	**5**	**6**
Räumlicher Ausschnitt			Klein	Klein	Klein	Klein	Groß	Groß
Anzahl der Zeitpunkte			1	3	1	3	1	3
	512							
	1.024							
	2.000							
	4.096							
	10.000	Ohne						
		Blosc ZSTD 9 [Deflate]						
		Blosc ZSTD 1 [Deflate (P2)]						
		Blosc LZ4 1 [LZW]						
		Blosc ZSTD 9 [LZW (P2)]						
		Blosc LZ4 9 [ZSTD]						
		LZ4 1 [ZSTD (P2)]						
		ZSTD 1						

Tabelle 4: Testkonfigurationen für verschiedene Chunking- und Kompressionsmethoden

Im Anschluss wird evaluiert, wie sich verschiedene Arten von Abfragen (räumlich, zeitlich, raumzeitlich) auf die Effizienz und Performance der Formate auswirken. Dabei wird insbesondere bei COG-Dateien

auch der Einfluss von Schnittgrenzen untersucht, d. h. von Abfragen, die zwischen mehrere Dateien liegen. Die Tests werden für jede Abfrageart auf allen verfügbaren Data Cubes durchgeführt. Für den kleinen Data Cube werden große Abfragebereiche nicht berücksichtigt, da die Datenmenge nicht ausreicht. Diese Untersuchung wird nur mit einer Kachelgröße von 10.000 × 10.000 Pixeln und ohne zusätzliche Kompression angewandt. Die Abfragetypen sind dabei folgende:

1. **Räumlich:** 2 DOPs an einem Zeitpunkt.
2. **Räumlich an Schnittstellen:** 2 DOPs an einem Zeitpunkt, wobei der Abfragebereich über 4 DOPs verteilt ist.
3. **Zeitlich:** 1 DOP über 2 Zeitpunkte.
4. **Kleiner raumzeitlicher Bereich:** 2 DOPs (verteilt auf 4 DOPs) über 3 Zeitpunkte.
5. **Großer raumzeitlicher Bereich:** 6 DOPs über 3 Zeitpunkte.

Die resultierenden Testfälle werden in Tabelle 5 dargestellt. Bei der Durchführung werden jeweils die Laufzeit, die Anzahl der HTTP-Requests, die Anzahl der Tasks und die übertragene Datenmenge erfasst. Die Bewertung erfolgt hinsichtlich der Anforderungen aus Kapitel 4.2. Die Ergebnisse werden durch die Ausgabe der Arrays und stichprobenartige RGB-Visualisierungen validiert. Für COG und Zarr werden identische Testbereiche und Parameter verwendet, um eine direkte Vergleichbarkeit zu gewährleisten.

	Räumlich	**Räumlich mit Schnitt**	**Zeitlich**	**Raumzeitlich (Klein)**	**Raumzeitlich (Groß)**
Testnummer	**7**	**8**	**9**	**10**	**11**
kleine Datenmenge					
Mittlere Datenmenge					
Große Datenmenge					

Tabelle 5: Testfälle für verschiedene raumzeitliche Abfragen verschiedener Zarr-Dateien

Teil 2: Raumzeitliche Analysen
Im zweiten Teil der Analyse wird das beschriebene Anwendungsszenario umgesetzt, das auf der Berechnung und Analyse des NDVI basiert, um Veränderungen der Vegetation über die Zeit zu erfassen und zu bewerten. Der NDVI dient dabei als Indikator für die Vegetationsentwicklung und ermöglicht Einblicke in die Vitalität des Naturraumes Harz und mögliche Auswirkungen von Schädlingen oder Umweltveränderungen. Er ist definiert als das Verhältnis der Differenz und Summe der Reflexionswerte im nahen Infrarotbereich (NIR) und im roten Bereich (RED) des elektromagnetischen Spektrums:

$$NDVI = \frac{(NIR - RED)}{(NIR + RED)} \tag{2}$$

Dieses Szenario dient als praktisches Beispiel zur Bewertung der Eignung der Formate für raumzeitliche Analysen. Dabei werden zwei raumzeitliche Analysen aus dem Kapitel 3.5 eingesetzt.

1. **Raumzeitliche Mittelwertberechnung:** Für die vorliegenden NDVI-Zeitpunkte wird der Durchschnittswert der Vegetationsentwicklung für jeden Zeitpunkt berechnet, um eine Übersicht der jährlichen Vegetationswerte zu erhalten. Außerdem wird der prozentuale Flächenanteil der Vegetation für diese Zeitpunkte bestimmt, um Waldverluste sichtbar zu machen. Dies entspricht einer deskriptiven Analyse, bei der die Vegetationsentwicklung an verschiedenen Zeitpunkten betrachtet wird, um Veränderungen sichtbar zu machen.
2. **Zeitliche Regressionsanalyse:** Im zweiten Schritt wird für jeden Pixel eine zeitliche Regressionsanalyse durchgeführt, um Trends in der Vegetationsentwicklung zu bestimmen. Dabei werden die NDVI-Werte für jeden Pixel über die verfügbaren Zeitpunkte hinweg verwendet, um eine lineare Regression zu berechnen. Diese Analyse liefert präzise Informationen über die langfristige Entwicklung der Vegetation für jeden Pixel. Da die Regression bestehende Datenpunkte nutzt, um fehlende Werte an nicht gemessenen Zeitpunkten zu schätzen. zählt dieses Verfahren zu den prädiktiven Analysen.

3. **Speicherung und pixelbasierte Modifizierung:** Abschließend werden die berechneten NDVI-Werte in den Datenformaten gespeichert. In einem weiteren Schritt erfolgt eine pixelbasierte Modifizierung, bei der gezielt einzelne Werte aktualisiert werden. Dieser Schritt dient dazu, die Flexibilität und Effizienz der Formate zu bewerten, insbesondere im Hinblick auf gezielte Änderungen einzelner Werte.

Die Berechnungen erfolgen für alle drei Datenmengen mit ausgewählten Partitionierungs- und Kompressionseinstellungen. Getestet wird ohne Kompression und mit dem ZSTD-Algorithmus bei einer Kachelgröße von 10.000 x 10.000 Pixeln. Zudem wird für jedes Format die optimale Partitionsgröße separat analysiert.

4.4 Verwendete Technologien

Für die Realisierung der Untersuchung stehen verschiedene Technologien zur Auswahl. Im Folgenden werden die eingesetzten Technologien vorgestellt und ihre Auswahl begründet. Für die Evaluierung cloud-optimierter Datenformate zur Speicherung und Analyse großer raumzeitlicher Rasterdaten wird eine skalierbare und flexible cloud-basierte Testumgebung aufgebaut. Diese Testumgebung wird gezielt so entworfen, dass sie den effizienten Zugriff auf große Datenmengen in der Cloud sowie deren parallele Verarbeitung ermöglicht. Dies ermöglicht es, die Datenverarbeitung nahe an der Datenspeicherung durchzuführen (siehe Prinzip „Code-to-Data “ in Kapitel 3.3) und die Vorteile der cloud-optimierten Datenformate zu nutzen und zu evaluieren.

Infrastrukturelle Technologien:
Die Wahl der verwendeten Technologien orientiert sich an wissenschaftlichen Standards und Best Practices im Bereich der cloud-basierten Geodatenverarbeitung. Ziel war eine cloud-native, Open-Source-Infrastruktur, die für die Verarbeitung von cloud-optimierten Rasterdatenformaten wie Zarr und COG optimiert ist. Die Konzeption orientierte sich dabei an etablierten Infrastrukturen für den Einsatz von Zarr in wissenschaftlichen Anwendungen. Sie basiert auf Empfehlungen der OGC zur Nutzung von Zarr für skalierbare, verteilte Datenverarbeitung (Open Geospatial Consortium, 2022b, o.S.) sowie auf Infrastrukturbeispielen aus der wissenschaftlichen Literatur wie in R. P. Abernathey et al., 2021, S. 33 beschrieben. Die Kombination von CODE-DE, OpenStack, Kubernetes und Dask bietet dabei eine fundierte Grundlage für die Evaluierung der Datenformate Zarr und COG in einem realen cloud-basierten Umfeld.

- **CODE-DE-Cloud:** Grundsätzlich wird die Untersuchung in der Cloud-Umgebung CODE-DE durchgeführt. Die CODE-DE Cloud ist ein zentraler Bestandteil der deutschen Geoinformationsstrategie und bietet einen einfachen, kostenfreien Zugang zu Fernerkundungsdaten sowie Cloud-Ressourcen für die Verarbeitung (Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), n. d., o.S.). Der Aufbau und Betrieb der Cloud erfolgte im Auftrag des Bundesministeriums für Digitales und Verkehr (BMDV) durch die Deutsche Raumfahrtagentur im DLR und der Firma CloudFerro S.A.. Diese Cloud-Infrastruktur ermöglicht es Nutzern, Virtuelle Maschinen (VMs) mit oder ohne GPU-Unterstützung, S3-kompatible Object Storage und eine Vielzahl an Tools zu verwenden, um ihre Prozesse zu verwalten und Daten effizient zu verarbeiten. Die Ressourcen werden in Form von Kontingenten vergeben und können flexibel für spezifische Anwendungsfälle genutzt werden. Zudem ist die CODE-DE Cloud durch die BSI-Zertifizierung und das C5-Testat als sichere Arbeitsumgebung für sensible Datenanwendungen zertifiziert. Diese Sicherheitsstandards machen die Cloud besonders geeignet für den Einsatz durch Behörden und öffentliche Institutionen. In der vorliegenden Untersuchung wird CODE-DE aufgrund der dort bereits verfügbaren kostenlosen Ressourcen des LGLN genutzt.

- **OpenStack:** Die CODE-DE-Cloud wird in der EO-Cloud von CloudFerro in Frankfurt/Deutschland gehostet. Die Einrichtung basiert auf der Open-Source-Cloud Computing-Software OpenStack (Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), n. d., o.S.). Diese Plattform stellt

eine IaaS-Lösung bereit, die es ermöglicht, Rechen-, Speicher- und Netzwerkressourcen in einem virtualisierten Cloud-Umfeld anzubieten. Die Bereitstellung und Steuerung dieser Ressourcen erfolgt über standardisierte OpenStack-APIs sowie ein webbasiertes Dashboard. Dabei fungiert OpenStack als Virtualisierungs- und Ressourcenmanagementschicht, die die Verwaltung und Zuteilung von Ressourcen wie virtuellen Maschinen (VMs), virtuellen CPUs (Virtual Central Processing Units (vCPUs)), RAM, Netzwerkverbindungen und Objektspeichern organisiert.

- **Kubernetes:** Kubernetes wird, wie in Kapitel 2.2 beschrieben, als Container-Orchestrierungsplattform genutzt, die den Betrieb und die Verwaltung der containerisierten Anwendungen in der CODE-DE-Cloud übernimmt. Die Nutzung von Kubernetes bietet sich an, da das Kubernetes-Cluster in der CODE-DE-Cloud bereits besteht und die vorhandene Infrastruktur für die Untersuchung optimal genutzt werden kann.

- **Daskhub:** Daskhub ist eine Plattform für parallele und verteilte Datenverarbeitung und spielt in dieser Untersuchung eine zentrale Rolle, da es die verteilte und parallele Datenverarbeitung durch eine effiziente Nutzung der Ressourcen innerhalb eines Kubernetes-Clusters erlaubt. In Kapitel 2.4 wurde das Dask -Framework bereits detaillierter erläutert. Dask integriert sich nahtlos mit *Xarray*, einem Python-Paket zur Analyse und Verarbeitung von multidimensionalen Daten und wird häufig in Kombination mit Zarr-Dateien verwendet (R. P. Abernathey et al., 2021, S. 33). Diese Eigenschaften machen Dask besonders geeignet, um große raumzeitliche Rasterdaten effizient zu verarbeiten, was auf einem lokalen System in diesem Umfang nicht möglich wäre.

- **JupyterHub:** JupyterHub dient als Benutzer- und Entwicklungsumgebung und stellt interaktive Jupyter-Notebooks für die Analyse und Visualisierung bereit (R. P. Abernathey et al., 2021, S. 31). In dieser Untersuchung wird JupyterHub als Schnittstelle genutzt, um auf die Daten, die Dask-Cluster und weitere Tools zuzugreifen.

Technologien für die Datenverarbeitung:
Für die Evaluation der cloud-optimierten Formate werden etablierte Geodatenverarbeitungsbibliotheken in Python verwendet. Dazu zählen neben Rasterverarbeitungsbibliotheken wie GDAL und Rasterio auch Bibliotheken für den Zugriff auf Objektspeicher wie S3fs und Boto3. Diese Auswahl stützt sich auf wissenschaftliche Literatur zur cloud-basierten Geodatenverarbeitung und stellt einen praxisnahen Anwendungsbezug sicher (Open Geospatial Consortium, 2022b, o.S.):

- **STAC-API:** Der offene Standard STAC bietet ein einheitliches Datenmodell und eine API-Spezifikation zur Katalogisierung und Organisation raumzeitlicher Geodaten im JavaScript Object Notation (JSON)-Format. Er ermöglicht die effiziente Suche, Verwaltung und Analyse großer Datenmengen aus verschiedenen Quellen (STAC Contributors, 2021, o.S.) und basiert auf einer hierarchischen Struktur mit den Hauptelementen Catalog, Collection, Item und Asset. Die STAC-API ergänzt dies durch eine RESTful-API, die eine dynamische Abfrage von STAC-Katalogen erlaubt. In dieser Untersuchung wird die STAC-API genutzt, um die DOPs effizient auszuwählen und zu laden. Die Nutzung erfolgt über die Bibliothek Pystac.

- **Xarray:** Xarray ist eine Open-Source-Python-Bibliothek, die für die Arbeit mit multidimensionalen Arrays und großen wissenschaftlichen Datensätzen konzipiert wurde (Hoyer und Hamman, 2017, S. 1). Zu den wichtigsten Merkmalen von Xarray gehören die labelbasierte Indizierung und Arithmetik, die Interoperabilität mit wissenschaftlichen Python-Paketen (z. B. Pandas, NumPy, Matplotlib) und die Unterstützung für Out-of-Core-Computing, das z. B. durch Dask, die Verarbeitung von Datensätzen ermöglicht, die nicht in den Speicher passen. Zudem bietet Xarray umfangreiche Optionen für Serialisierung und Ein-/Ausgabe sowie fortschrittliche Werkzeuge zur Manipulation multidimensionaler Daten wie Groupby und Resampling. Xarray bietet zwei verschiedene Datenstrukturen:

  - **Data-Array:** Dies entspricht einem beschrifteten, mehrdimensionalen Array. Es kombiniert die eigentlichen Daten mit Koordinaten, Attributen und Metadaten, die die Daten beschreiben wie z. B. Orts- und Zeitkoordinaten.

– **Dataset:** Ein Dataset ist eine Sammlung von DataArrays, die gemeinsame Dimensionen teilen können. Es kann als eine Art Container für mehrere beschriftete Data-Arrays betrachtet werden.

Rioxarray ist eine Erweiterung der Python-Bibliothek Xarray, die speziell für die Arbeit mit räumlichen Geodaten entwickelt wurde (Wu, 2024, o.S.). Sie bietet eine Schnittstelle zwischen Xarray und der populären Bibliothek Rasterio, die für das Lesen, Schreiben und Bearbeiten von Rasterdaten verwendet wird. Die Bibliothek baut auf NumPy auf und fügt zusätzliche Funktionalitäten hinzu, was sie besonders geeignet für die Analyse von Geodaten, Klimadaten und anderen wissenschaftlichen Datensätzen macht.

4.5 Architektur

Die Architektur zur Evaluierung cloud-optimierter Rasterdatenformate wie in Abbildung 18 dargestellt umfasst eine skalierbare, containerisierte Umgebung zur Verarbeitung und Analyse von Rasterdaten in der Cloud. OpenStack fungiert dabei als Hardware-Abstraktionsschicht. Diese Plattform stellt eine IaaS-Lösung bereit, die die Verwaltung und Zuteilung von Ressourcen wie VMs, vCPUs, RAM, Netzwerkverbindungen sowie den Objektspeicher organisiert. OpenStack bildet somit wie in Abbildung 18 dargestellt die unterste Infrastrukturebene. Das LGLN nutzt Kontingente der CODE-DE-Cloud. Dabei stehen in der CODE-DE-Cloud Ressourcen in Form von vier VMs mit jeweils 16 vCPUs und 128 GB RAM sowie zwei VMs mit jeweils vier vCPUs und 16 GB RAM zur Verfügung. Auf den von OpenStack bereitgestellten VMs wird ein Kubernetes-Cluster implementiert. Dies stellt die Orchestrierungsebene dar (siehe Abbildung 18).Auf Applikationsebene sind dann ein DaskHub und ein JupyterHub eingerichtet (siehe Abbildung 18). Der DaskHub stellt die Umgebung für verteilte Datenverarbeitung zur Verfügung und fungiert auf dieser Ebene als eine Datenverarbeitungs- und Orchestrierungsplattform innerhalb von Kubernetes. Mit dem Dask Gateway ist die dynamische Erstellung von Dask-Cluster-Instanzen für verteilte Workloads möglich. Der JupyterHub dient als Umgebung für das interaktive Arbeiten und stellt Benutzern eine zentrale Plattform für die Erstellung und Ausführung von Jupyter-Notebooks bereit. Somit resultiert ein mehrschichtiges und hochskalierbares Architekturmodell. Die vertikale Skalierung geschieht auf der OpenStack-Ebene. Die Kapazität der einzelnen VMs kann über OpenStack angepasst werden, indem beispielsweise vCPUs oder RAM hinzugefügt werden. Die horizontale Skalierung geschieht auf Kubernetes-Ebene, indem neue Pods erstellt werden. Die Verfügbarkeit von Ressourcen auf dieser Ebene hängt jedoch von der Anzahl der zugrunde liegenden VMs ab. Die konkrete Funktionsweise dieser Architektur ist in Abbildung 19 dargestellt.

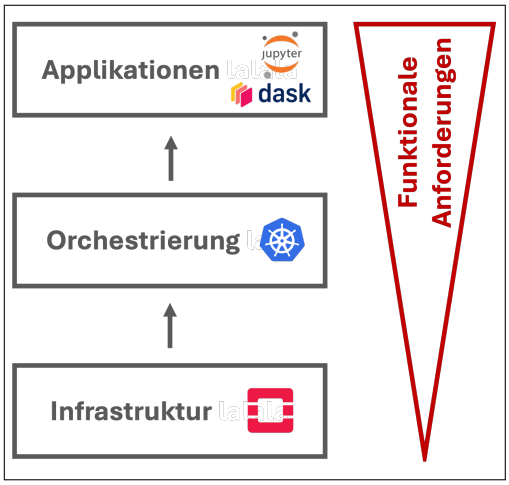

Abbildung 18: Schichtenmodell der Architektur (Eigene Darstellung)

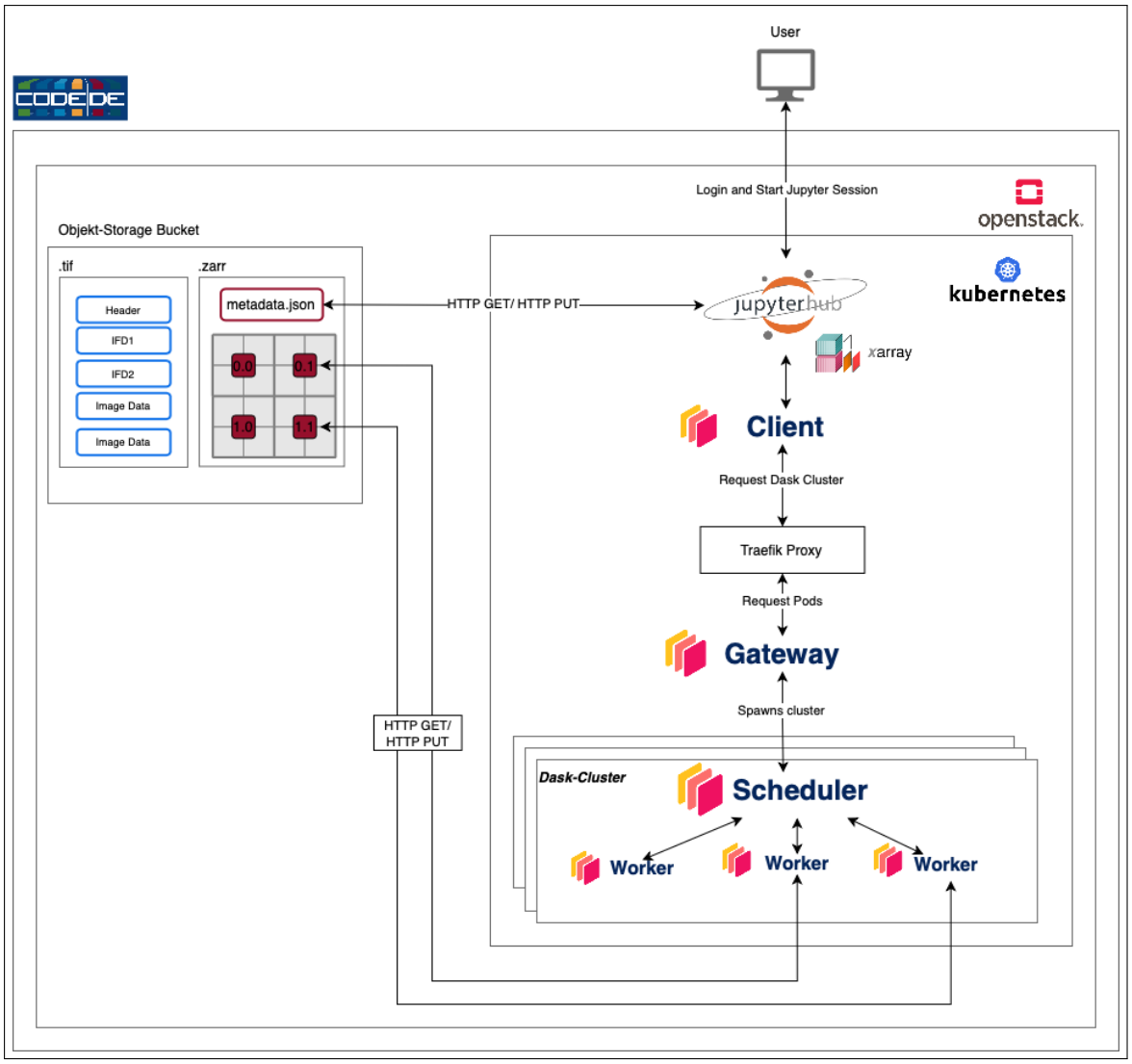

Abbildung 19: Architektur (Eigene Darstellung)

Die Speicherung und Analyse der cloud-optimierten Rasterdatenformate erfolgt über einen Object-Storage-Bucket, der in der **CODE-DE-Cloud** mithilfe von **OpenStack** gehostet wird. Parallel dazu wird in der CODE-DE-Umgebung ein **Kubernetes-Cluster** betrieben, das vom LGLN verwaltet wird. Dieses Cluster greift über OpenStack auf die bereitgestellten Rechenressourcen zu. Innerhalb des Kubernetes-Clusters laufen sowohl ein **JupyterHub** als auch ein **DaskHub**, die die zentrale Schnittstelle für Benutzer bilden. Über diese Infrastruktur können HTTP-Requests zur Abfrage oder Speicherung von Daten an den **Object-Storage-Bucket** gesendet werden. Die Metadaten der COG- und Zarr-Dateien können direkt innerhalb der Entwicklungsumgebung in Jupyter gelesen oder geschrieben werden. Dies ermöglicht eine nahtlose Interaktion mit den Datenformaten. Partielle Zugriffe sowie die darauf aufbauenden Verarbeitungsprozesse nutzen das Dask-Gateway als Kommunikationsschnittstelle. Hierbei koordiniert das Gateway die Dask-Worker. Diese führen die parallele Datenverarbeitung aus, indem sie HTTP-Requests an den Object-Storage-Bucket senden, um die erforderlichen Datenblöcke zu laden (siehe Abbildung 19).

Das vorliegende Sequenzdiagramm in Abbildung 20 beschreibt den konkreten Ablauf der Interaktion eines Users mit der Dask-Umgebung und dem JupyterHub. Der Prozess beginnt mit der Anmeldung des Users bei JupyterHub. Die Authentifizierung erfolgt durch einen Traefik Proxy, welcher den Zugang zum Kubernetes-Cluster kontrolliert. Nach erfolgreichem Login und Starten eines Jupyter-Servers wird ein Pod im Kubernetes-Cluster initiiert, der das Jupyter Notebook bereitstellt und als Entwicklungsumgebung für die Evaluierung der Rasterdatenformate dient (Rückgabe: „Authentication successful“).

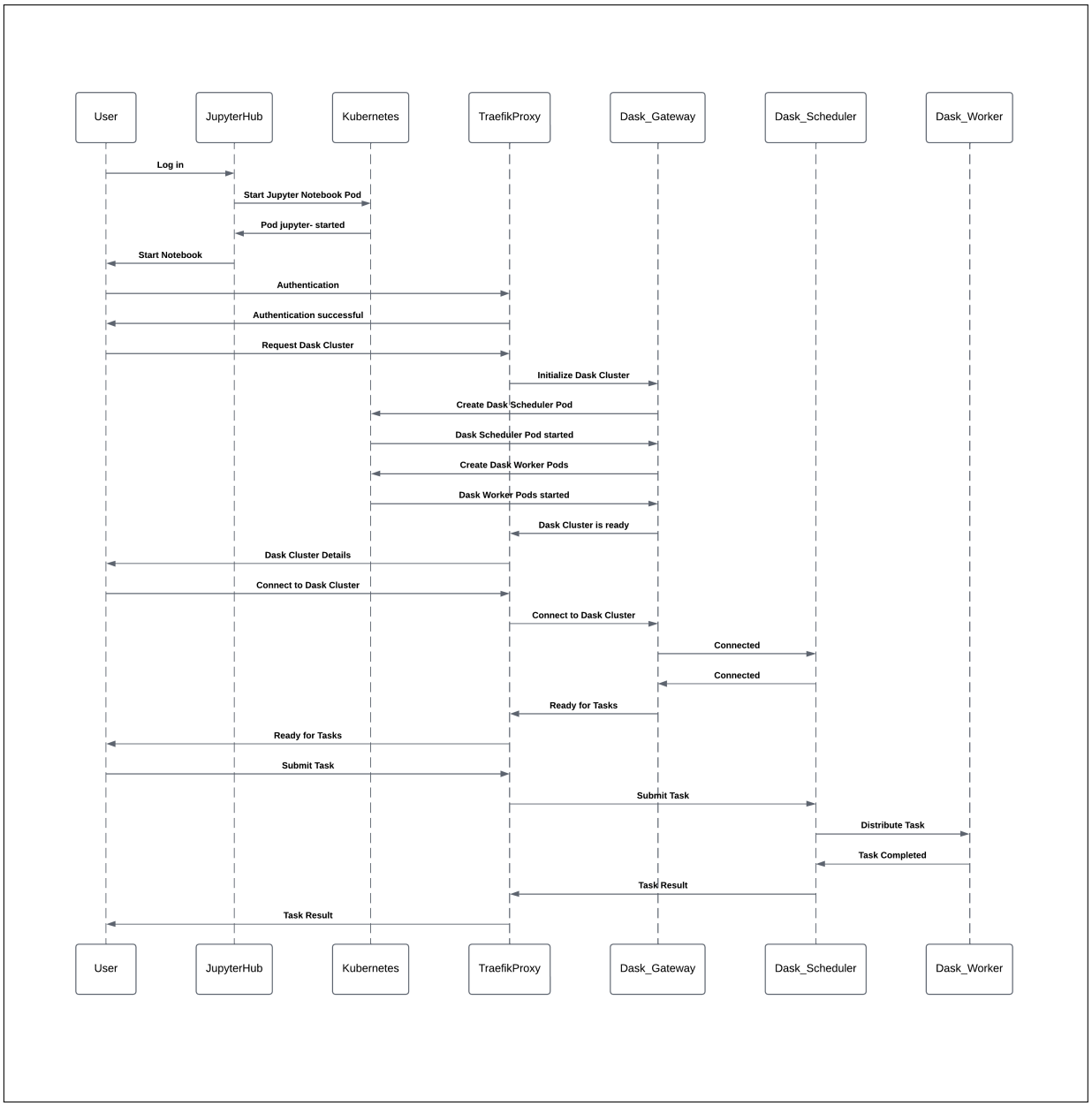

Abbildung 20: Sequenzdiagramm Dask (Eigene Darstellung)

Im nächsten Schritt fordert der User die Bereitstellung eines Dask-Clusters an (Eingabe: „Request Dask Cluster“). Diese Anfrage wird durch das Dask-Gateway verarbeitet, welches die Initialisierung des Clusters übernimmt. Der Zugriff auf das Dask Gateway wird dabei durch den Dask-Traefik Proxy authentifiziert. Der Dask Gateway-Controller hat die Rechte, in Kubernetes Pods zu spawnen. Bei erfolgreicher Authentifizierung startet das Dask Gateway über Kubernetes einen Dask-Cluster, bestehend aus einem Scheduler-Pod und mehreren Worker-Pods. Innerhalb des Kubernetes-Clusters werden die Dask-Pods (Scheduler und Worker) beliebig auf verschiedene Knoten verteilt, abhängig von den verfügbaren Ressourcen. Abbildung 21 zeigt eine mögliche, schematische Darstellung der Pod-Verteilung innerhalb des Clusters. Die vier Knoten enthalten einen Pod für den Dask-Scheduler und mehrere Pods für die Dask-Worker. Die Worker sind auf verschiedene Knoten verteilt. Zudem existiert ein Pod für den Traefik-Proxy von Dask sowie jeweils ein Pod für den JupyterHub und das gestartete Jupyter Notebook.

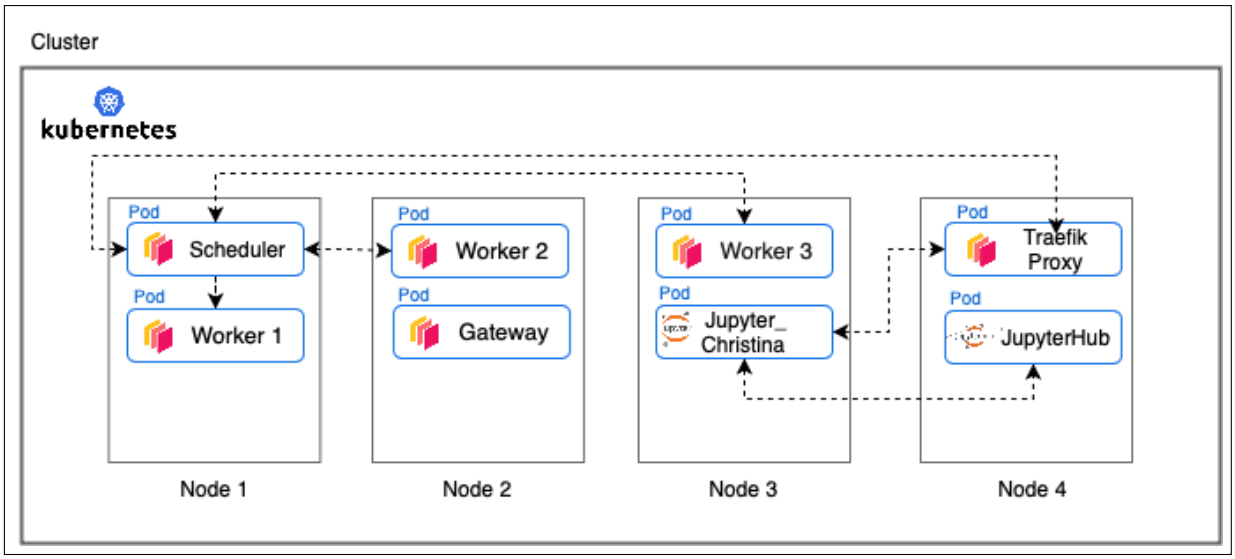

Abbildung 21: Schematische Darstellung des Kubernetes-Clusters (Eigene Darstellung)

Sobald das Cluster bereit ist, wird im Notebook ein Cluster-Objekt zurückgegeben (Rückgabe: „Dask Cluster Details“). Um Verarbeitungsprozesse auf diesem Cluster ausführen zu können, wird über den Traefik Proxy und das Dask-Gateway eine Verbindung zum Scheduler aufgebaut. In der Phase der Aufgabenausführung übermittelt der User eine Aufgabe an das Dask-Cluster (Eingabe: „Submit Task“). Dabei sendet der Dask-Client die Berechnungsaufgaben als Tasks über den Traefik Proxy an den Scheduler, der diese an die Worker-Pods verteilt. Sobald die Worker die Aufgaben abgeschlossen haben, werden die Ergebnisse an den Scheduler zurückübermittelt und anschließend an das Jupyter-Notebook zurückgegeben (Rückgabe: „Task Result“).

5 Theoretischer Vergleich von COG und Zarr

Aus den Bemühungen der letzten Jahre für ein cloud-optimiertes Datenformat für Rasterdaten sind besonders die Formate COG und Zarr hervorgegangen. Beide Formate bieten spezifische Vorteile und Herausforderungen, die in Bezug auf die Anforderungen an moderne Geodatenstrukturen differenziert zu bewerten sind. Während COG auf dem bewährten GeoTIFF-Format basiert und speziell für den schnellen Lesezugriff in Cloud-Umgebungen optimiert ist, setzt Zarr auf eine chunkbasierte und multidimensionale Struktur, die auf verteilte und parallele Verarbeitung ausgelegt ist.

Im Folgenden werden die beiden Formate im Hinblick auf die Anforderungen moderner raumzeitlicher Datenverarbeitung verglichen. Dabei werden grundlegenden Konzepte und technischen Hintergründe im Hinblick auf die Anforderungen aus Kapitel 4.2 diskutiert. Dies umfasst neben technischen Kriterien, wie dem partiellen Zugriff, Modifizierbarkeit und der Kompression, auch die Einschätzung der Interoperabilität sowie die Unterstützung der Zeitdimension und raumzeitlichen Abfragemechanismen. Das Ziel ist es, die Architektur, die Speicherstrukturen und die Zugriffsmechanismen der Formate zu untersuchen und zentrale Gemeinsamkeiten und Unterschiede herauszuarbeiten. Die Identifizierung dieser Aspekte erlaubt Rückschlüsse auf die Eignung der Formate für die Speicherung und Analyse raumzeitlicher Rasterdaten. Zudem bilden die theoretischen Erkenntnisse eine wesentliche Grundlage für die nachfolgende praktische Evaluierung und beleuchten mögliche Grenzen und Herausforderungen.

Partitionierung und Parallelverarbeitung
Eine zentrale Anforderung an Datenformate für raumzeitliche Rasterdaten ist die effiziente Verwaltung und der gezielte Zugriff auf bestimmte räumliche und zeitliche Teilbereiche großer Datensätze. Sowohl COG als auch Zarr implementieren hierfür verschiedene Mechanismen der Partitionierung, um partielle Zugriffe und Parallelverarbeitung zu ermöglichen.

COG nutzt eine tile-basierte Struktur, die in Kapitel 3.4.1 detailliert beschrieben ist. Diese ermöglicht durch Byte-Serving und HTTP-Range-Requests effizienten partiellen Zugriff und parallele Lesevorgänge, was besonders für große, räumlich ausgedehnte Datenbestände vorteilhaft ist. Um den Inhalt einer Kachel partiell abzufragen, sind stehts mindestens zwei HTTP-Requests notwendig. Mit einem Request wird der Headers zur Positionsbestimmung der Kachel abgefragt und mit dem anderen Request erfolgt die Abfrage der tatsächlichen Daten. Die parallele Verarbeitung erfolgt, indem die Tiles, die jeweils einen räumlichen Ausschnitt des Bildes darstellen, auf verschiedene Worker verteilt werden. Jeder Worker liest und verarbeitet mittels HTTP-Requests die ihm zugewiesenen Tiles unabhängig von den anderen, wodurch parallele Berechnungen über verschiedene Bildbereiche hinweg ermöglicht werden. Ein wesentlicher Unterschied zu Zarr liegt in der zentralisierten Organisation der Datei. Alle Kacheln werden in einer einzigen Datei gespeichert, die gezielt ausgelesen werden kann, jedoch keine parallelen Schreiboperationen erlaubt. Dies wird im Abschnitt Modifizierbarkeit näher behandelt.

Im Gegensatz zu COG setzt Zarr auf eine chunk-basierte Organisation, die die parallele Verarbeitung und das parallele Schreiben von Daten nativ unterstützt. Während COG durch Byte-Serving optimierte Zugriffe ermöglicht, basiert Zarr auf einer dezentralen Speicherung der Chunks in einer hierarchischen Ordnerstruktur. Diese Struktur ist besonders für verteilte Anwendungen vorteilhaft, kann jedoch bei großen Datensätzen eine sehr hohe Anzahl von Dateien erzeugen, was für manche Speichertechnologien ineffizient ist (Miles, 2019, o.S.). Wie in Kapitel 3.4.2 erläutert, erfolgt der partielle Zugriff in Zarr durch die präzise Bestimmung des Dateinamens der relevanten Chunks. Auch hier sind mindestens zwei Requests erforderlich: einer für die Metadatenabfrage und einer für den Abruf der Chunk-Datei. Zarr unterstützt parallele Lese- und Schreibvorgänge durch seine dezentrale Chunk-Struktur. Jeder Chunk wird unabhängig gespeichert und kann somit von verschiedenen Prozessen gleichzeitig gelesen oder geschrieben werden, ohne Blockierungen oder Sperren zu verursachen. Wie in Abbildung 22 dargestellt, greifen mehrere Prozesse parallel auf unterschiedliche Chunk-Dateien zu. Die parallele Verarbeitung eines Data Cubes erfolgt, indem die Chunks basierend auf ihrer Größe, der Anzahl verfügbarer Rechenkerne und dem verfügbaren Speicher auf die Worker verteilt werden. In einem dreidimensionalen Data Cube mit den Dimensionen x, y und Zeit könnte der Würfel beispielsweise entlang der räumlichen Dimension (x und y) aufgeteilt werden, sodass jeder Chunk einen räumlichen Ausschnitt über alle Zeitpunkte enthält. Jeder Worker verarbeitet dann gleichzeitig einen Chunk und erzeugt ein Teilergebnis. Nachdem alle Teilergebnisse erstellt wurden, werden sie zu einem finalen Ergebnis zusammengefügt. Das parallele Schreiben einzelner Chunks funktioniert nach einem ähnlichen Prinzip und wird im Abschnitt Modifizierbarkeit näher erläutert.

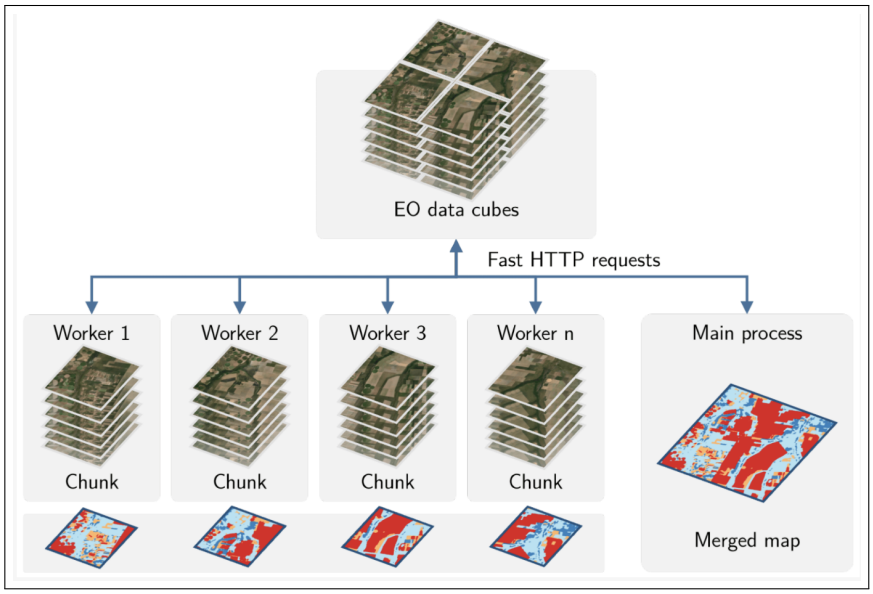

Abbildung 22: Parallele Verarbeitung mit einem Data Cube (Simoes et al., 2021, S. 10)

Speicherreduktion:
Für die effiziente Handhabung großer raumzeitlicher Rasterdaten ist es notwendig, den Speicherbedarf und Übertragungskosten zu reduzieren. Beide Datenformate unterstützen verschiedene Kompressionsmethoden, um den Speicherbedarf zu reduzieren. COG haben eine interne Komprimierung der Datei, was bedeutet, dass die internen Tiles in einem GeoTIFF bereits komprimiert sind. Da COG eine Tiling-Struktur verwendet, wird jede Kachel unabhängig komprimiert (Cloud-Native Geospatial Foundation, 2023, o.S.). Dabei werden verlustfreie und verlustbehaftete Kompressionsmethoden wie Deflate, LZW oder JPEG unterstützt. Deflate und LZW sind verlustfreie Kompressionsmethoden, die die Dateigröße ohne Qualitätsverlust reduzieren, während JPEG als verlustbehaftetes Verfahren verwendet wird, um höhere Kompressionsraten, z. B. für die Visualisierung, zu erreichen. Die Wahl der Kompressionsmethode kann je nach Anwendungsfall angepasst werden. Weiterhin können Nullwerte in COG als No-Data gekennzeichnet werden. Diese Markierung zeigt Analysetools wie GIS an, dass diese Pixel keine gültigen Daten enthalten. COG unterstützt jedoch keine dedizierten, leeren Tiles, weshalb auch die „Nodata"-Pixel Speicherplatz beanspruchen.

Zarr geht einen anderen Ansatz zur Speicherreduktion, der speziell auf mehrdimensionale und große Datensätze ausgelegt ist. Zarr unterstützt eine Vielzahl von Kompressionsalgorithmen, darunter Blosc, Zlib, LZ4 und Zstd. Diese Algorithmen können nach Bedarf als verlustfreie oder verlustbehaftete Kompression angewendet werden. Ein großer Vorteil von Zarr ist die Unterstützung sogenannter „empty chunks". Dies sind Datenblöcke, die ausschließlich Nodata-Werte enthalten. Indem leere Chunks, die nur diese Nodata-Werte enthalten nicht mit gespeichert werden, zeigt sich diese Technik besonders vorteilhaft für heterogene Datensätze, bei denen Bereiche des Data Cubes keine Informationen enthalten.

Skalierbarkeit:
Die Fähigkeit eines Datenformats, wachsende Datenmengen, neue Dimensionen oder Variablen flexibel zu handhaben, ist ein zentraler Aspekt moderner Geodatenverarbeitung. Formal basiert COG auf GeoTIFF, das ausdrücklich von TIFF Version 6 abhängig und auf 4 GB-Dateien beschränkt ist. Diese Einschränkung wird von den Entwicklern durch die Unterstützung von BigTIFF gemildert (Open Geospatial Consortium, 2022a, o.S.). Das GeoTIFF-Format ist ursprünglich nur für zweidimensionale Rasterdaten konzipiert und unterstützt Erweiterungen nur bedingt. Zusätzliche Dimensionen können durch die Speicherung in separaten Dateien oder durch die Verwendung von Bildbändern innerhalb einer Datei abgebildet werden. Während dies für kleinere, überschaubare Datensätze funktioniert, wird die Verwaltung bei wachsender Datenmenge oder zusätzlichen Dimensionen schnell komplex und unübersichtlich. Zarr unterstützt nativ zusätzliche Dimensionen und Variablen, ohne die bestehende Datenstruktur neu organisieren zu müssen. Neue Zeitpunkte, räumliche Abschnitte oder Variablen können einfach hinzugefügt werden, was Zarr besonders geeignet für wachsende Datensätze macht. Da Zarr keine festen Dateigrößenbegrenzungen hat, können selbst große Datenmengen problemlos verwaltet werden. Jedoch steigt mit zunehmender Datenmenge potentiell auch die Anzahl der Chunk-Dateien. Zudem gibt es erste Ansätze, die darauf abzielen, Zarr für eine größere Vielfalt von Datentypen zu erweitern. Dabei wird die flexible Struktur von Zarr genutzt, um auch große, komplexe Vektordaten effizient als Data Cube zu speichern und zu verarbeiten (Marshall, 2024, o.S.).

Modifizierbarkeit:
Wie in Kapitel 3.4.1 beschrieben, ist die Struktur von COG-Dateien primär auf effiziente Lesevorgänge ausgelegt und weniger für Modifikationen. Jede Änderung einer Kachel erfordert eine Anpassung der IFDs und der Dateistruktur, was bei großen Dateien erheblichen Speicher- und Rechenaufwand bedeutet. Wird ein Tile verschoben oder eingefügt, müssen die IFDs aktualisiert und die Metadaten komplett neu generiert werden. Aufgrund der festen Struktur kann dies eine Anpassung der Tile-Reihenfolge in der Datei erforderlich machen. Paralleles Schreiben erfordert, dass jeder Prozess auf den gesamten IFD-Bereich zugreift, um neue Byte-Positionen und Größen zu referenzieren. Da alle Kacheln in einer einzigen Datei organisiert sind, müssten Sperren verwendet werden, um Konsistenz zu gewährleisten. In der Praxis wird eine COG-Datei bei Änderungen oft komplett neu erstellt.

Im Gegensatz dazu unterstützt Zarr, wie in Kapitel 3.4.2 erläutert, durch seine dezentrale Chunk-Struktur paralleles Schreiben und Modifikationen auf Chunk-Ebene. Alle Änderungen des Arrays, die keine Neuordnung erfordern, werden effizient unterstützt. Wird eine Dimension verkleinert, entfernt Zarr alle Chunks außerhalb der neuen Array-Form. Zum Beispiel löscht eine Reduktion eines Arrays mit Dimensionen x, y und Zeit auf zwei Zeitschritte alle Dateien mit Endungen wie x.x.2 oder höher. Zum Vergrößern nutzt Zarr eine Append-Funktion, um Daten an jede Achse anzuhängen. Wird ein ein elfter Zeitschritt zu einem Array mit 10 Zeitschritten hinzugefügt, erstellt Zarr einfach eine neue Datei, z. B. „2.3.10“ für den Chunk bei x=2, y=3 und dem neuen Zeitschritt. Bestehende Dateien bleiben unverändert, was paralleles Schreiben erleichtert.

Das Einfügen neuer Dimensionen oder Daten zwischen bestehenden Chunks ist jedoch technisch aufwändig und ressourcenintensiv. Für solche Operationen, die eine Neustrukturierung der Daten erfordern, ist Zarr nicht optimiert. Eine neue Dimension, wie spektrale Informationen, erfordert die Umstrukturierung des gesamten Datensatzes. Alle Dateinamen und die Ordnerstruktur müssten angepasst werden, z. B. von „2.3.1“ zu „2.3.1.0“ für eine zusätzliche Dimension „Band“. Ebenso würde das Einfügen neuer Daten zwischen Zeitschritten alle nachfolgenden Chunks verschieben und umbenennen. In der Praxis wird daher meist eine neue Datei erstellt oder die bestehende Überschrieben. Oftmals ist es ratsam am Anfang ein leeres Array entsprechend zu dimensionieren und die Werte nach und nach zu ergänzen, falls die Zielgröße bekannt ist.

Ein wichtiger Aspekt bei der Nutzung von Zarr-Dateien ist jedoch der Mehrbenutzerbetrieb. Gleichzeitige Änderungen am selben Chunk können zu Inkonsistenzen führen, da Zarr keine ACID-Transaktionen unterstützt (Zarr Developers, 2024, S. 29). Da jeder Chunk eine separate Datei darstellt, können mehrere Benutzer jedoch parallel auf verschiedene Chunks zugreifen und diese ändern. Um die Datenkonsistenz bei parallelen Schreibzugriffen im Mehrbenutzerbetrieb zu gewährleisten, ist es notwendig, zusätzliche Softwarelösungen einzusetzen, die Schreibprozesse koordinieren und absichern können. Die Zarr-Community hat in diesem Zusammenhang Lösungen wie Arraylake und IceChunk entwickelt. IceChunk fügt ACID-Transaktionen und Versionierung hinzu und macht parallele Schreibvorgänge möglich (R. Abernathey, 2024, o.S.). Arraylake erweitert dies als umfassende Data-Lake-Plattform mit Funktionen wie rollenbasiertem Zugriff, Metadatensuche und Datenkatalogen, um große wissenschaftliche Datensätze effizient zu verwalten und eigene Infrastrukturen zu vermeiden.

Raumzeitliche Datenstrukturen:
Moderne Geodatenanwendungen erfordern zunehmend die Fähigkeit, Daten nicht nur räumlich, sondern auch zeitlich zu strukturieren und zu analysieren. Dies ermöglicht es, Veränderungen über Zeit und Raum hinweg zu analysieren und Trends oder Muster zu erkennen. Dafür müssen die Formate sowohl eine räumliche Dimension, eine zeitliche Dimension als auch nicht-räumliche Attribute und Metadaten speichern können.

COG ist für die Speicherung von zweidimensionalen Rasterbildern entwickelt worden und daher primär für räumliche Datenstrukturen optimiert. Die Datenstruktur basiert auf dem in Kapitel 3.1 vorgestellten Snapshot-Modell, bei dem jede Aufnahme einen Zeitpunkt unabhängig von anderen darstellt. Zeit- und andere Dimensionen werden deshalb nur eingeschränkt unterstützt. Zur Speicherung der Zeitdimension gibt es technisch zwei Optionen:

1. **Speicherung einzelner Dateien:** Jede Datei enthält einen räumlichen Ausschnitt für einen spezifischen Zeitpunkt. Der Zeitpunkt wird in den Metadaten hinterlegt. Diese Struktur ist konsistent mit den ARD-Prinzipien und erlaubt eine einfache Verwaltung einzelner Dateien. Es entsteht jedoch keine zusammenhängende Zeitdimension und eine zusätzliche Dateiverwaltung ist notwendig. Dies geschieht in der Praxis häufig über Katalogsysteme wie STAC oder spezifische Datenmanagementsysteme. Die Verarbeitung bleibt flexibel, jedoch wird die Anzahl der Dateien bei großen Zeitserien schnell unübersichtlich.

2. **Speicherung über Bänder:** Zeitpunkte werden sequentiell in den Bildbändern gespeichert, wobei jedes Band einen spezifischen Zeitpunkt repräsentiert. Dies reduziert die Anzahl der Dateien, erfordert jedoch eine präzise Dokumentation in den Metadaten, um die Zuordnung der Bänder zu gewährleisten.

Metadaten werden direkt in der Datei gespeichert, was schnellen Zugriff und automatische Berechnung von Bildstatistiken ermöglicht. Dies macht COG ideal für ARD, die als Momentaufnahmen betrachtet und analysiert werden. Die Integration weiterer Dimensionen (z. B. Höhenebenen oder zusätzliche Spektralbänder) ist jedoch technisch limitiert und wird mit zunehmender Anzahl an Bändern unpraktisch.

Zarr ist speziell für die Speicherung mehrdimensionaler, wissenschaftlicher Datensätze entwickelt und basiert auf dem Data Cube-Modell, das die Integration von Dimensionen wie Raum, Zeit und Spektralbändern in einer kohärenten Struktur ermöglicht. Im Gegensatz zu COG, das auf dem Snapshot-Modell basiert, können mit Zarr große Zeitseriendaten effizient gespeichert und flexibel abgefragt werden, ohne separate Dateien für jeden Zeitschritt erstellen zu müssen. Zur Speicherung der Zeitdimension gibt es mehrere Möglichkeiten:

1. **Speicherung in einem Array:** Ein Zarr-Array kann beliebig viele Dimensionen speichern, was es ermöglicht, sowohl räumliche als auch zeitliche Dimensionen abzubilden. Diese Struktur erleichtert die Abfrage und Verarbeitung der Daten entlang beliebiger Kombinationen und Reihenfolgen von Dimensionen. Die Zeitdimension kann in Zarr direkt als eine eigene Dimension definiert werden. Dies ermöglicht es, große Zeitseriendaten effizient zu speichern und schnell auf bestimmte Zeitpunkte oder Zeitintervalle zuzugreifen, ohne dass separate Dateien für jeden Zeitschritt notwendig sind.

2. **Speicherung einzelner Arrays:** Alternativ können einzelne Arrays für jeden Zeitschritt oder gruppiert nach der Zeitdimension (z. B. nach Jahr) gespeichert werden. Dabei muss jedes Array separat abgefragt werden, sodass keine integrative Zeitdimension entsteht.

Langzeituntersuchungen oder Trendanalysen, die mehrere Zeitpunkte umfassen, profitieren erheblich von der integrativen Struktur von Zarr. Im Vergleich dazu führt eine solche Analyse mit COG zu einer Vielzahl von HTTP-Range-Requests, da jede Datei separat abgefragt werden muss. Diese Fragmentierung ist ineffizient, wenn viele Dateien gemeinsam verarbeitet werden sollen. Häufig müssen COG-Dateien vorab zu einem aggregierten Datensatz zusammengeführt werden, was zusätzlichen Speicher- und Rechenaufwand erfordert. Die Flexibilität von Zarr bringt Herausforderungen mit sich. Ohne interne Konsistenzprüfung können unregelmäßige Zeitserien gespeichert werden, was die Analyse erschwert und eine genaue Kenntnis der Datenstruktur erfordert. Ein Beispiel wäre eine Zarr-Datei mit orientierten Luftbildern des LGLN, die aufgrund des Befliegungsturnus mit kurzen zeitlichen Sequenzen für einen Bereich vorliegen und gleichzeitig jahrelange Lücken aufweisen, deren korrekte Handhabung vom Endnutzer ein Verständnis der vorhandenen Datenstruktur verlangt.

Im Gegensatz zu COG verfolgt Zarr einen integrativen Ansatz, bei dem Daten und deren Organisation innerhalb des Formats selbst abgebildet werden. Zarr fungiert somit als eine Art Katalog, der multidimensionale Daten und Metadaten in einer einzigen kohärenten Struktur integriert. Die JSON-basierte Struktur erlaubt es, Metadaten auf Array-, Gruppen- oder Dateiebene zu speichern. Da bei einem Data Cube oft mehrere Dateien zu einem großen Array zusammengefügt werden, gehen die individuellen Metadaten der Einzelbilder verloren. Zudem erfordert Zarr eine manuelle Berechnung von Statistiken. Durch die Integration der Daten und Metadaten direkt im Format benötigt Zarr jedoch keine separate Verwaltungsinstanz. Ein Nachteil dieser engen Integration ist, dass Änderungen an der Datenorganisation bei Zarr aufwendig sein können, da die gesamte Datei oder Struktur neu aufgebaut werden muss. Im Vergleich dazu erfordern COG-Datensammlungen zusätzliche Infrastrukturen wie Katalogsysteme (STAC), um zeitliche und räumliche Abfragen zu ermöglichen. Diese Trennung von Daten und Metadaten erhöht die Flexibilität, birgt jedoch das Risiko von Inkonsistenzen, wenn Änderungen nicht synchron gehalten werden. Ohne Katalog ist der Zugriff auf COG-Datensammlungen mühsam und unstrukturiert.

Standardisierung und Interoperabilität:
Eine wichtige Anforderung an moderne Datenformate ist die Standardisierung, um eine nahtlose Integration in bestehende Systeme und Workflows zu ermöglichen. Dies fördert die Interoperabilität und ermöglicht die einfache Nutzung in GIS sowie den Datenaustausch zwischen verschiedenen Plattformen unter Einhaltung geographischer Standards (vgl Anforderung 2).

COG basiert auf dem weit verbreiteten GeoTIFF-Standard, der in der Geoinformation etabliert ist und von nahezu allen relevanten Softwarelösungen unterstützt wird. Es handelt sich um ein offenes Format und einen anerkannten Standard des OGC. COG wird nativ von nahezu allen GIS wie QGIS und ArcGIS und Programmbibliotheken wie GDAL und Rasterio unterstützt (Chris Holmes et. al., 2024, o.S.). Auch Web-Visualisierungstools wie Leaflet und OpenLayers können COG-Dateien problemlos darstellen. Durch die Erweiterung des GeoTIFF-Formats zur Unterstützung von HTTP-Range-Requests kann COG nicht nur in Desktop-GIS, sondern auch direkt im Browser und in Cloud-Umgebungen genutzt werden. Diese breite Unterstützung ermöglicht die Nutzung von COG ohne zusätzliche Anpassungen oder Konvertierungen in einer Vielzahl von Plattformen vom GeoServer bis zur Google Earth Engine. Zusätzlich unterstützt COG standardisierte Metadaten, gängige Koordinatenreferenzsystem (CRS) und OGC-Standards, was es zu einem universellen und vielseitigen Format in der Geodatenverarbeitung macht.

Zarr, ursprünglich für multidimensionale wissenschaftliche Datensätze entwickelt, ist ein flexibles Format mit wachsender Akzeptanz. Es wurde als OGC Community Standard vorgeschlagen, hat jedoch noch nicht den vollständigen Standardisierungsprozess durchlaufen. Die Zarr-Spezifikation wird jedoch aktiv weiterentwickelt und es gibt Bestrebungen, Zarr in wissenschaftlichen und industriellen Workflows als De-facto-Standard für mehrdimensionale Daten zu etablieren (Open Geospatial Consortium, 2022c, S. 3). Zarr erlaubt die Speicherung von Metadaten und CRS, erfordert jedoch, dass diese Informationen bei der Erstellung explizit definiert werden. Dies umfasst auch Aspekte wie Datentypen und die Konsistenz der gesamten Datenstruktur. Zarr unterstützt sowohl projizierte als auch geographische Koordinaten. Ohne eine native Unterstützung standardisierter CRS-Spezifikationen wie EPSG oder PROJ liegt die Verantwortung für korrekte Metadaten jedoch vollständig bei der datenproduzierenden Instanz.

Obwohl Zarr in wissenschaftlichen Bibliotheken wie Xarray und Dask gut integriert ist, wird es bisher von klassischen GIS wie QGIS oder ArcGIS und Geodatenverarbeitungstools wie Rasterio nur begrenzt unterstützt. Diese Einschränkung ergibt sich aus dem multidimensionalen Charakter von Zarr und der fehlenden festen Vorgaben zur Speicherung geodatenrelevanter Informationen. Um Zarr-Daten in GIS zu verwenden, sind GDAL-Versionen ab 3.8 sowie spezifische Kompressionsalgorithmen wie BLOSC erforderlich. Overviews werden nicht unterstützt, da Zarr-Arrays separat geladen werden. Enthält ein Array mehrere Dimensionen oder Zeitpunkte, wird in GIS derzeit nur die oberste Ebene des Data Cubes dargestellt. Darüber hinaus werden Rasterbilder in gängigen Formaten wie GeoTIFF in GIS auch ohne definiertes Koordinatensystem meist maßstabsgetreu dargestellt, nur ohne räumlichen Bezug. Bei Zarr hingegen ist dies nicht der Fall. Ohne korrekt definiertes CRS fehlt jegliche räumliche Orientierung, was die Nutzung in GIS stark einschränkt. Unterschiedliche Bibliotheken wie GDAL oder Xarray erzeugen zudem abweichende Datenstrukturen, was die Konsistenz erschwert. Bei der Nutzung mit GDAL entsteht beispielsweise eine flachere Hierarchie, ohne separate Ordner für Dimensionen wie Zeit oder spektrale Bänder (siehe Abbildung 23). Im Gegensatz dazu generiert Xarray standardmäßig eine multidimensionale Struktur mit separaten Unterordnern (siehe Abbildung 24). Es ist auch möglich, Zarr-Daten in einer vollständig flachen Hierarchie zu speichern, bei der die Chunk-Dateien direkt auf der obersten Ebene abgelegt werden. Solche flachen Hierarchien sind auch mit älteren GDAL-Versionen (vor 3.8) kompatibel.

Im Gegensatz zu COG können Zarr-Datensätze nicht direkt über eine URL im Browser visualisiert werden. Zarr erfordert spezielle Tools, um die Datenstruktur zu interpretieren, was die Nutzung in Web-Umgebungen erschwert. Insgesamt hängt die Interoperabilität von Zarr derzeit stark von der Einhaltung von Standardkonventionen durch die Nutzenden ab und davon, wie sich zukünftige Standards entwickeln.

Langfristig müssen auch Institutionen wie das OGC und die AdV solche mehrdimensionalen Strukturen und Data Cubes in ihre bestehenden Standards aufnehmen, um eine breitere Unterstützung und Integration zu ermöglichen.

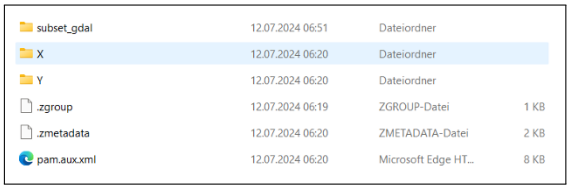

Abbildung 23: Erstellung einer Zarr-Datei mit GDAL (Eigene Darstellung)

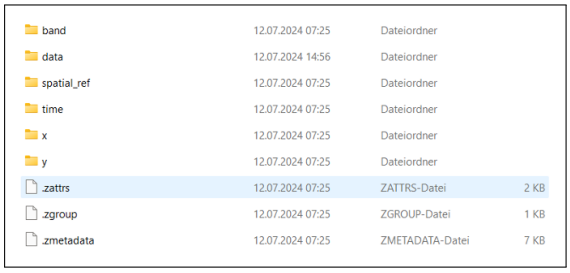

Abbildung 24: Erstellung einer Zarr-Datei mit Xarray (Eigene Darstellung)

Übersichten:
Visualisierung ist der Schlüssel zur Erforschung und zum Verständnis von räumlichen Daten. Ein wichtiger Aspekt ist dabei die Fähigkeit, Daten effizient auf unterschiedlichen Zoomstufen bereitzustellen. Dies ermöglicht eine schnelle Navigation und Visualisierung, ohne die gesamte Datenmenge in der höchsten Auflösung laden zu müssen. Der Umfang von Geodaten macht es schwierig, diese Daten sofort bereitzustellen. Dies führte zur Entwicklung vorgenerierter statischer Kartenkacheln in Form von Overviews. COG unterstützt die Erstellung von Overviews nativ. Dazu nutzt COG das Konzept der pyramidalen Überblicke (Overviews), die in Kapitel 3.4.1 vorgestellt wurden. Diese Hierarchie von Bildern in absteigender Auflösung ermöglicht eine schnelle Navigation und Visualisierung, da GIS automatisch die passende Zoomstufe anahnd der IFDs erkennen und laden. Die Erzeugung der Overviews geschieht bei der Erstellung eines COG i.d.R. automatisch. Standardmäßig wird die Anzahl der optimalen Übersichtsebenen basierend auf der Größe des Datensatzes und der internen Kachelgröße bestimmt. Dabei sollte der Overview nicht kleiner sein als die interne Kachelgröße. Die Auflösung wird dabei in mehreren Stufen schrittweise reduziert, wobei jede Stufe einer Zweierpotenz entspricht (cogeotiff, n. d., o.S.). Änderungen an der Anzahl der Overviews oder der Auflösung können manuell bei der Erstellung vorgenommen werden. Die Zarr-V2-Spezifikation hingegen hat kein festes Konzept von pyramidalen Überblicken, aber die flexible, multidimensionale Chunk-Struktur erlaubt eine ähnliche Funktionalität. Überblicke können z. B. über die Gruppierung berücksichtigt werden, wie das Toolkit von Carbonplans zur Generierung von Bildpyramiden verdeutlicht (Freeman et al., 2021, o.S.). Die flexible Chunk-Struktur erlaubt so die manuelle Erstellung von Bildpyramiden oder niedrig aufgelösten Arrays. Diese können nicht nur räumlich, sondern auch für weitere Dimensionen wie Zeit gespeichert werden, was zusätzliche Flexibilität bietet. Da Zarr-Overviews manuell erstellt und in GIS nicht automatisch erkannt werden, ist die Handhabung für Zoomstufen im Vergleich zu COG deutlich komplexer.

Zusammenfassung:
Zarr bietet eine modernere, skalierbare Lösung, die effizienten, parallelen Zugriff und Anpassungsfähigkeit für wachsende Datensätze ermöglicht. COG hingegen ist aufgrund seiner Standardisierung, Interoperabilität und Optimierung für GIS-Visualisierungen besser geeignet, wenn hohe Kompatibilität und schnelle, räumliche Abfragen priorisiert werden. Die Ergebnisse sind in der nachfolgenden Tabelle zusammengefasst:

Kriterium	COG	ZARR
Partitionierung	Tile-basiert (zentral in einer Datei)	Chunk-basiert (verteilte Dateien)
Partieller Zugriff	HTTP-Range-Requests	Indexierung über Chunk-Dateinamen
Parallelität	Paralleles Lesen kein paralleles Schreiben	Paralleles Lesen und Schreiben
Metadaten	Ja (in Datei selbst - Header)	Ja (Json)
Interoperabilität	Sehr gut	Gut - mittel
Kompression	Ja	Ja
Datentyp	Raster	Mehrdimensionale Array
Übersichten	nur räumlich	Nicht nativ
Datenmodell	ARD (Snapshot-Modell)	ARD/Data Cube-Modell
Zeitdimension	Zeitstempel in Metadaten Zeitpunkte als Bildbänder	Integrierte Zeitdimension
Unterstützung partieller raumzeitlicher Abfragen	partiell nur räumlich	Zeitlich und räumlich
Modifizierbarkeit	Nein	Bedingt (Änderungen auf Chunk-Ebene ohne Neustruktuierung)
Nullwerte	Kennzeichnnung als NO-Data	Ausschluss leerer Chunks

Tabelle 6: Vergleich zwischen COG und ZARR

6 Implementierung

Unter Berücksichtigung der Erkenntnisse aus dem theoretischen Vergleich wird in diesem Kapitel die praktische Umsetzung des zuvor entwickelten Konzeptes beschrieben. Es umfasst die Einrichtung der Entwicklungsumgebung und die Analyse der Dask-Parameter, um eine leistungsfähige Infrastruktur für die Verarbeitung und Analyse der Daten zu schaffen. Anschließend werden Speicherstrategien für die Formate COG und Zarr entwickelt und die Daten in die Cloud hochgeladen, wobei spezifische Herausforderungen und Lösungsansätze für jedes Format dargestellt werden. Ein weiterer Schwerpunkt liegt auf der Implementierung raumzeitlicher Abfragen und Analysen. Hier werden die Funktionalitäten der Formate getestet, um ihre Eignung für praktische Anwendungen zu bewerten.

6.1 Einrichtung der Entwicklungsumgebung

Um die in Kapitel 4.5 beschriebene Architektur umzusetzen, war es erforderlich, eine geeignete Entwicklungsumgebung einzurichten. Innerhalb des LGLN stand hierfür die CODE-DE-Cloud Umgebungen zur Verfügung. Diese Umgebung bietet die Möglichkeit, ein bestehendes Kubernetes-Cluster zu nutzen, auf denen Dask installiert werden kann, um eine parallele und verteilte Datenverarbeitung zu ermöglichen. Die Installation von Dask im Kubernetes Cluster erfolgt über ein Helm Chart. Die Installation erfolgt analog zu dem in Kapitel 2.2 beschriebenen Vorgehen. Dabei wurde sich an der Anleitung aus der Dask-Dokumentation Jim Crist-Harif, 2021, o.S. orientiert. Hierzu wird das Helm Chart heruntergeladen und die *values.yaml*-Datei angepasst. Diese Anpassungen umfassten die Optionen für die Dask-Worker, die Einstellungen für den JupyterHub sowie die Entfernung der Authentifizierungskonfiguration und der Node Affinity. Die values.yaml-Konfiguration gliedert sich wie folgt:

- **JupyterHub-Einstellungen:** Hier werden Netzwerkeinstellungen für den JupyterHub-Dienst implementiert, um eine Integration mit dem Dask Gateway sicherzustellen. Die Netzwerkregeln erlauben beispielsweise nur den Zugriff auf Pods mit dem Label app.kubernetes.io/name: dask-gateway über den Port 8000, wodurch eine sichere Verbindung für die Dask-Kommunikation gewährleistet ist. Jeder Instanz erhält standardmäßig 5 GB Speicher.
- **Dask Gateway-Einstellungen:** Das Dask Gateway ermöglicht die flexible Steuerung von Cluster-Ressourcen direkt aus JupyterHub. Es werden Optionen für die Skalierbarkeit der Dask-Worker

konfiguriert, darunter Einstellungen für die Anzahl der CPU-Kerne (zwischen 1 und 4) und den Arbeitsspeicher pro Worker (zwischen 1 und 32 GB). Zudem erlaubt die Konfiguration die Auswahl des Docker-Images, das für die Dask-Worker verwendet wird und legt Timeout-Einstellungen fest.

- **Dask-Kubernetes:** In diesem Abschnitt wird Dask-Kubernetes deaktiviert, was darauf hinweist, dass die Cluster-Orchestrierung allein über den Dask Gateway erfolgt.

Die Konfiguration beschreibt die gewünschten Ressourcen wie Deployments, Pods und Services. Mit dem folgenden Befehl wird der DaskHub eingerichtet:

helm install –repo https://helm.dask.org –create-namespace -n daskhub –generate-name daskhub

Helm rendert diese Datei basierend auf den Werten in der values.yaml-Datei und sendet sie dann an den Kubernetes-API-Server. Der API-Server sorgt dafür, dass die notwendigen Ressourcen erstellt werden, z. B. Pods, die die Anwendung ausführen und Services, die den Netzwerkzugriff zu den Pods ermöglichen. Dask-Clients werden in den Jupyter Notebooks von JupyterHub ausgeführt. Um sicherzustellen, dass für den Scheduler und die Worker dieselbe Umgebung verwendet wird, müssen diese als Gateway-Option angegeben werden. Aus diesem Grund muss später bei der Dask-Cluster-Initialisierung das Image als Cluster-Option mit übergeben werden (vgl. Listing 2). Zusätzlich wird die Authentifizierungskonfiguration und die Node Affinity entfernt, damit die Dask-Worker nicht auf GPU-Knoten im Kubernetes-Cluster erstellt werden (siehe Listing 1).

```
extraPodConfig:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: nvidia.com/gpu.present
                operator: NotIn
                values:
                - 'true'
```

Listing 1: Dask-Authentifizierungskonfiguration

Um im Jupyter-Notebook ein Dask-Cluster zu erstellen, wird zunächst ein Dask Gateway-Objekt initialisiert und eine Cluster-Konfiguration festgelegt (siehe Listing 2). Diese Konfiguration spezifiziert den Arbeitsspeicher, die Anzahl der Kerne pro Worker und das Container-Image. Die Erstellung eines neuen Clusters erfolgt durch den Aufruf von *gateway.new_cluster()*. Der Dask Gateway-Controller hat die Rechte im Kubernetes Pods zu spawnen. Bei erfolgreicher Authentifizierung startet das Dask Gateway über Kubernetes einen Dask-Cluster, bestehend aus einem Scheduler-Pod und mehreren Worker-Pods. Die Anzahl der Worker-Pods kann dynamisch durch den Befehl *cluster.scale()* angepasst werden. Um Dask richtig verwenden zu können, muss zunächst sichergestellt werden, dass der Client (in diesem Fall das Jupyter-Notebook) sowie der Scheduler und die Worker alle mit der gleichen oder einer kompatiblen Version von Dask arbeiten. Da im Default-Image für die Worker im Helmchart eine andere Version von Dask angegeben ist, muss bei der Erstellung eines Clusters die Option *Image* für die Scheduler/Worker auf den Wert „pangeo/base-notebook:2024.01.15“ eingestellt werden. Sobald das Cluster bereit ist, erhält der Benutzer im Notebook ein Cluster-Objekt zur weiteren Verwendung zurück. Über den Befehl *register_plugin()* können zusätzliche Python-Bibliotheken wie Rioxarray, Pystac, Boto3 und Rio-cogeo im Cluster installiert werden, um die spezifischen Anforderungen der Datenverarbeitung zu erfüllen (siehe Listing 2). Damit ist das Cluster einsatzbereit für die folgenden Untersuchungen.

```
def createGatewayCluster(num_worker, worker_memory, worker_cores):
gateway = Gateway()
options = gateway.cluster_options()
options.worker_memory = worker_memory
```

```
    options.worker_cores = worker_cores
    options.image = "pangeo/base-notebook:2024.01.15"
    cluster = gateway.new_cluster(cluster_options=options)
    cluster.scale(num_worker)
    client = cluster.get_client()
    packages = ['rioxarray', 'pystac', 'boto3', 'rio_cogeo']
    for p in packages:
        plugin = PipInstall(packages=[p], pip_options=["--upgrade"])
        client.register_plugin(plugin)
        print('plugin installiert')
    return client, gateway, cluster
```

Listing 2: Create Dask-Cluster

6.2 Analyse der Dask-Parameter

Die effiziente Verarbeitung und Analyse großer raumzeitlicher Rasterdatensätze in der Cloud-Umgebung erfordert eine sorgfältige Abstimmung der verwendeten Werkzeuge und Parameter. Dabei spielt Dask als Framework für parallele und verteilte Berechnungen eine entscheidende Rolle. Eine sorgfältige Justierung der Dask-Parameter ist notwendig, um eine optimale Balance zwischen Performance und Ressourcennutzung zu erreichen. Zur Erstellung eines Clusters in Dask können drei Parameter verwendet werden:

- Anzahl der Worker
- Threads pro Worker
- Arbeitsspeicher pro Worker

Bereits bei ersten Tests zeigte sich, dass die Konfiguration von Dask einen signifikanten Einfluss sowohl auf die Performance des Dateiuploads (Erstellen und Speichern der Rasterdaten) als auch auf die Performance der Ladevorgänge (Zugriff und Analyse der gespeicherten Daten) hat. Daher ist es unerlässlich, den Einfluss der einzelnen Parameter vorab zu evaluieren. Da die Auswirkungen der Parameter beim Uploadprozess deutlicher hervortraten, wird die nachfolgende Analyse auf diesen Aspekt fokussiert. Für die Ladevorgänge werden die gleichen Ressourcen und Parameter verwendet, sodass die Ergebnisse in beiden Prozessen konsistent bleiben. Im Folgenden werden die Parameter für die CodeDE-Umgebung anhand des Uploadprozesses optimiert, um sowohl die Leistung zu maximieren als auch die Ressourcenbeschränkungen der CodeDE-Cloud aus Kapitel 6.1 einzuhalten. Dies verhindert Engpässe und gewährleistet einen reibungslosen Ablauf auch bei ungünstige Kachelgrößen. Zur Konfiguration müssen die Optionen im Helm-Chart um den folgenden Code ergänzt werden:

```
def option_handler(options):
    return {
        "worker_cores": options.worker_cores,
        "worker_memory": "%fG" % options.worker_memory,
        "image": options.image,
    }
    c.Backend.cluster_options = Options(
        Integer("worker_cores", 2, min=1, max=4, label="Worker Cores"),
        Float("worker_memory", 4, min=1, max=8,
        label="Worker Memory(GiB)"),
        String("image", default="daskgateway/dask-gateway:latest",
        label="Image"),
        handler=option_handler)
```

Listing 3: Worker-Optionen im Helm-Chart

Die Standard-Einstellungen für die Cluster-Optionen werden aus der Dask-Dokumentation übernommen (Jim Crist-Harif, 2021, o.S.). Pro Worker muss mindestens ein Thread existieren. Insgesamt ist die Anzahl der Threads auf maximal vier beschränkt. Der Arbeitsspeicher pro Worker kann bis maximal acht

GB hochskaliert werden. Die Anzahl der Worker ist technisch nicht beschränkt, hängt aber von den zur Verfügung stehenden Ressourcen in der Cloud ab (siehe Kapitel 6.1). Die Justierung der Parameter erfolgt über Testmessungen. Dabei werden die Auswirkungen der verschiedenen Einstellungen auf die Performance des Dateiuploads und der Ladevorgänge beider Dateiformate untersucht. Ziel ist es, durch die Analyse eine Konfiguration zu finden, die das optimale Verhältnis aus Verarbeitungsgeschwindigkeit und Ressourcenverbrauch darstellt. Jede Messung wird zweimal durchgeführt, um Ausreißer auszuschließen. Während der Tests werden das Dask-Dashboard, Konsolenausgaben und Protokolle überwacht, um Probleme wie Netzwerkfehler oder ineffizientes Ressourcenmanagement zu erkennen.

Die Boxplot-Analyse in Abbildung 25 zeigt, dass die Abweichungen zwischen den verschiedenen Messungen mit zunehmender Datengröße bei Zarr-Dateien steigt. Insgesamt treten Abweichungen von bis zu 30 Sekunden auf. Die durchschnittlichen Abweichungen betragen je nach Datenmenge zwischen 8 und 17 Sekunden. Ein ähnliches Verhalten ist auch beim Upload von COG-Dateien festzustellen. Diese Zunahme der Variabilität lässt sich vermutlich auf Faktoren wie Netzwerkschwankungen und Systemauslastung zurückführen, die bei größeren Datenmengen stärker ins Gewicht fallen. Bei mehrfachen Wiederholungen können sogar Schwankungen bis zu einer Minute auftreten. Angesichts dieser Beobachtungen ist es insbesondere bei größeren Datenmengen wichtig, Untersuchungen mehrmals durchzuführen, um verlässliche Durchschnittswerte zu erhalten und Ausreißer zu identifizieren. Die Ladezeit der Ausgangsraster war in allen Parameter-Konstellationen sehr konstant, weshalb sie in den folgenden Abschnitten vernachlässigt wird.

Das Radardiagramm 26 gibt einen ersten Überblick über die Korrelationen der Parameter zur Uploaddauer mit einer Ausgangsdatenmenge von 25 DOPs. Jeder Parameter wird auf der Skala von -1 bis 1 dargestellt, wobei -1 auf eine starke negative Korrelation mit der Uploaddauer hinweist. Je stärker die negative Korrelation, desto stärker wirkt sich der jeweilige Parameter auf die Verkürzung der Uploadzeit aus. Aus dem Diagramm geht hervor, dass die Uploadzeit mit einer Korrelation von 75% maßgeblich von der Anzahl der Worker abhängt. Das bedeutet, dass die Uploadzeit sich mit zunehmender Anzahl an Workern signifikant verringert. Der Einfluss der Threads und des Arbeitsspeichers pro Worker auf die Uploadzeit ist vergleichsweise geringer, zeigt aber ebenfalls eine negative Korrelation. Eine höhere Anzahl an Cores und mehr Arbeitsspeicher pro Worker verbessert die Parallelisierung der Arbeitsschritte innerhalb jedes Workers, was die Uploadzeit weiter reduziert. Dennoch zeigen die Ergebnisse, dass es insgesamt vorteilhafter ist, mehr Worker mit weniger Speicher zu nutzen, als wenige Worker mit mehr Speicher. Die ersten Testläufe zeigten bereits, dass eine detailliertere Analyse der Parameter notwendig ist, da einige Durchläufe beim Upload von COG- und Zarr-Dateien fehlschlugen. Ein besseres Verständnis dieser Parameter soll ermöglichen, fundierte Entscheidungen über die optimale Dask-Konfiguration für die weiteren Untersuchungen zu treffen.

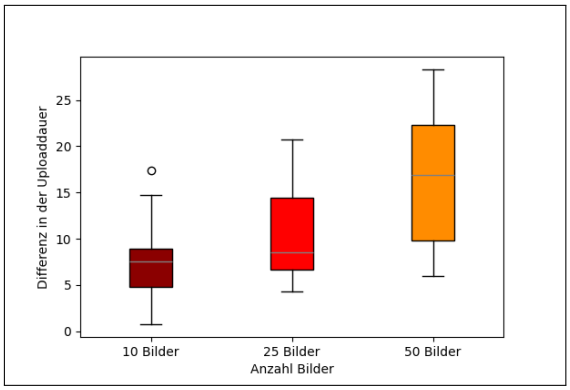

Abbildung 25: Schwankungen in der Uploadzeit von Zarr-Dateien

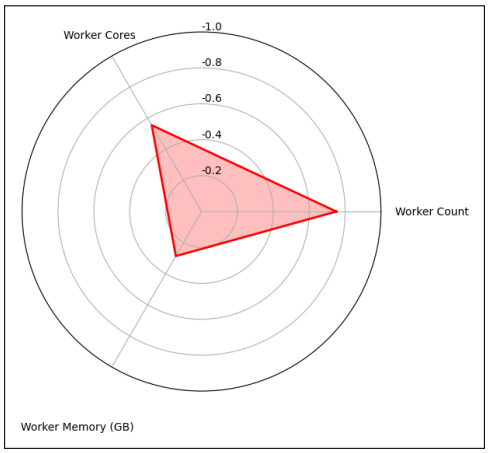

Abbildung 26: Einfluss der Dask-Parameter auf die Uploadzeit einer Zarr-Datei

Arbeitsspeicher pro Worker:
Die Analyse des Arbeitsspeichers zeigt, dass ein unzureichender Arbeitsspeicher dazu führen kann, dass der Upload fehlschlägt. In diesen Fällen war die Anzahl der Tasks und die Uploaddauer signifikant erhöht, da einige Worker aufgrund von Speichermangel und Overhead-Abstürzen mehrmals neu gestartet wurden. Zu häufige Worker-Abstürze führen letztlich zu einem Abbruch. Interessanterweise benötigen die Worker trotz optimal gewählter Chunk-Größen, die den Best Practices von Dask entsprechen, mehr Arbeitsspeicher als zur Verfügung steht (Anaconda, Inc. and Contributors, 2018c, o.S.). Zarr teilt große Arrays in kleinere, handhabbare Chunks auf. Dadurch kann jeder Chunk separat in die Cloud hochgeladen werden. Der Dask-Scheduler koordiniert dabei die parallele Verarbeitung und den Upload der unabhängigen Chunks. Bei der Verarbeitung von COGs wird ein ähnlicher Ansatz verfolgt. Unter Vernachlässigung anderer Einflussfaktoren wie der Netzwerkbandbreite und I/O-Geschwindigkeit könnte ausgehend von einer Chunk-Größe von 500 MB und einer Speicherkapazität von zwei GB pro Worker theoretisch angenommen werden, dass jeder Worker bis zu vier Chunks simultan verarbeiten und hochladen kann. Obwohl die zu verarbeitenden Chunks klein genug sind, um mehrere gleichzeitig pro Worker zu verarbeiten, werden einzelne Worker in Dask überladen. Das Worker-Diagramm auf dem Dask-Dashboard zeigt durch orangefarbene Balken eine Warnung, dass sich der Speicher dem Limit nähert und rot bedeutet, dass der Worker runtergefahren wird (siehe Abbildung 27).

Bei genauerer Betrachtung des Dask-Dashboards fällt jedoch auf, dass der Arbeitsspeicher die meiste Zeit über gar nicht ausgenutzt wird. Auch mit unterschiedlichen Chunk-Größen ändert sich dieses Verhalten nicht. Im Durchschnitt verwendet ein Worker rund ein bis zwei GB. Insbesondere mit wachsenden Datenmengen zeigt sich, dass einige Worker überladen werden statt die Daten auf zusätzliche Worker zu verteilen. Dies führt zu einer ineffizienten Ressourcennutzung, da der Worker mehr Speicher benötigt, als im Durchschnitt erforderlich wäre. Mehr noch führt ein zu geringer Arbeitsspeicher zum Abbruch des Uploads.

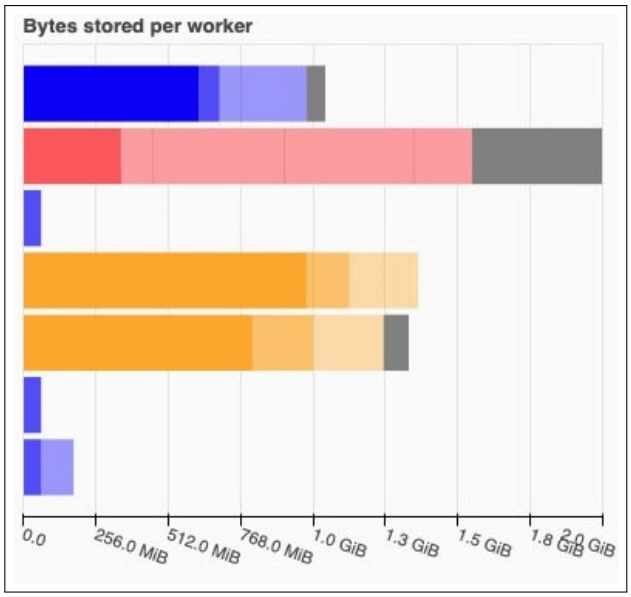

Abbildung 27: Ausschnitt aus dem Dask-Dashboard beim Upload einer Zarr-Datei

Die Überlastung einzelner Worker hängt nicht unbedingt mit der Arraygröße oder der Anordnung und Größe der Dimensionen des Arrays zusammen. Vielmehr ist das Problem auf ineffizientes Scheduling und „Root Task Overproduction“ zurückzuführen und ist ein bekanntes Problem mit Dask (Jetter, 2023, o.S.). Dabei erzeugt Dask zu viele Aufgaben (Root Tasks) gleichzeitig, was dazu führt, dass Daten länger als notwendig im Speicher gehalten werden. Während der Tests zeigte sich, dass die Anzahl der tatsächlichen Tasks deutlich höher ist als die Anzahl der Chunks. Ein mehrdimensionales Array mit 50 Bildern und 960 Chunks führte zu ca. 3000 Tasks beim Upload als Zarr-Datei und zu ca. 2.000 Tasks beim Upload von COG-Dateien. Dies liegt daran, dass Dask für jede Operation (Laden, Verarbeiten,

Speichern) eigene Tasks erstellt. Prozesse wie Rechunking, Concatenating und Broadcasting erhöhen die Komplexität des Task-Graphs weiter. Dask nutzt dabei ein Priorisierungssystem (dask.order), das die Ausführungsreihenfolge der Tasks bestimmt. Dabei werden Root-Tasks oft zu hoch priorisiert (Joseph, 2023, o.S.). Diese Probleme werden in mehreren GitHub-Diskussionen intensiv thematisiert und es wird an Verbesserungen im Scheduling-Verhalten gearbeitet, um diese Speicherengpässe zu minimieren (Jetter, 2023, o.S.). Zukünftige Versionen sollen eine optimierte Task-Reihenfolge implementieren, um den Speicherverbrauch bei der Verarbeitung großer Datenmengen zu reduzieren. Für die Untersuchungen führen die beschriebenen Herausforderungen zu einer gewissen Unvorhersehbarkeit in der Speichernutzung. Die Verarbeitung der Chunks und die Arbeitsspeichernutzung pro Worker lassen sich nicht präzise steuern. Aufgrund dessen muss der Parameter „Worker-Memory“ entsprechend hoch angesetzt werden, um einen sicheren Upload zu gewährleisten. Dieser wird daher auf zehn GB festgelegt.

Threads pro Worker:
Der Parameter „Worker-Cores“ hat einen positiven Einfluss auf die Uploadzeit, während die Ladezeit und das Zusammenfügen der Rasterbilder wie bereits beim Parameter „Worker-Memory“ relativ stabil bleiben. Dabei zeigt sich, dass die größere Reduktion von einem auf zwei Threads erfolgt. Allerdings muss die Anzahl der Threads pro Worker sorgfältig abgewogen werden, um ein Gleichgewicht zwischen Verarbeitungsleistung und Ressourcennutzung herzustellen. Eine Verdopplung der Threads halbiert die mögliche Anzahl der Worker bei gleichen Gesamtressourcen. In Abbildung 26 ist zu erkennen, dass der Parameter „Worker Count“ einen stärkeren Einfluss auf die Uploadzeit der Dateien hat als der Parameter „Worker Cores“. Konkret bedeutet dies, dass beispielsweise eine Konfiguration mit acht Workern und je einem Core ($\sim$ 105 Sek.) effizienter ist als vier Worker mit je zwei Cores ($\sim$ 143 Sek.). Basierend auf diesen Erkenntnissen wird die Anzahl der Threads pro Worker auf Zwei festgelegt.

Anzahl der Worker:
Die Erhöhung der Worker-Anzahl beschleunigt die Verarbeitungszeit und verkürzt signifikant die Uploadzeit der Rasterdateien. Das Verhalten der Upload-Zeiten zeigt eine hyperbolische Form, die an das Amdahl'sche Gesetz aus Kapitel 2.3 erinnert und typisch für parallele Verarbeitungsprozesse in Dask ist (Bohm und Beranek, 2020, S. 8). Es beschreibt, dass die Reduktion der Uploadzeit durch eine Erhöhung der Worker zunächst stark ist, jedoch mit zunehmender Anzahl an Workern weniger Effizienzgewinne erzielt werden, da der maximale Geschwindigkeitszuwachs durch den seriellen Anteil der Aufgabe begrenzt wird (Kuzmiakova, 2022, S. 129). Die Uploadzeiten lassen sich somit durch eine vereinfachte Form des Amdahl'schen Gesetzes beschreiben: $f(x) = 1/x$. Besonders bei einer Erhöhung von 2 auf 12 Worker reduzierte sich die Uploadzeit von COG- und Zarr-Dateien für 25 Rasterbilder um über 75%. Für kleinere Datenmengen ist der Nutzen zusätzlicher Worker jedoch geringer, da der Overhead schneller zum Tragen kommt. Dies zeigt sich in flacheren Kurvenverläufen und entspricht dem Prinzip des abnehmenden Grenznutzens (Diminishing Returns) aus Kapitel 2.3.

Abbildung 28 und 29 zeigen die Verbesserung der Uploadgeschwindigkeit in Abhängigkeit der Worker-Anzahl in Sekunden für COG- und Zarr-Dateien. Die Steigung der Uploadgeschwindigkeit weist eine logarithmische Form auf, wie es bei parallelen Verarbeitungsprozessen üblich ist (Ristov et al., 2016, S. 894). Dies ist insbesondere an den R^2-Werten ersichtlich, die zeigen, dass die Modelle die Daten gut beschreiben. Die vertikalen gestrichelten Linien markieren den sogenannten „Ellenbogenpunkt“, an dem die Steigerung der Worker-Anzahl nur noch geringere Verbesserungen der Uploadzeit bringt. Ab diesem Punkt nehmen die Effekte des Overheads zu. Der Ellenbogenpunkt stellt hierbei eine wertvolle Metrik dar, um Kosten zu minimieren und gleichzeitig die maximale Performance zu erreichen. Der Ellenbogenpunkt liegt für Zarr-Dateien zwischen 4 und 12 Workern, je nach Datenmenge, während er bei COG-Dateien zwischen 6 und 10 Workern liegt.

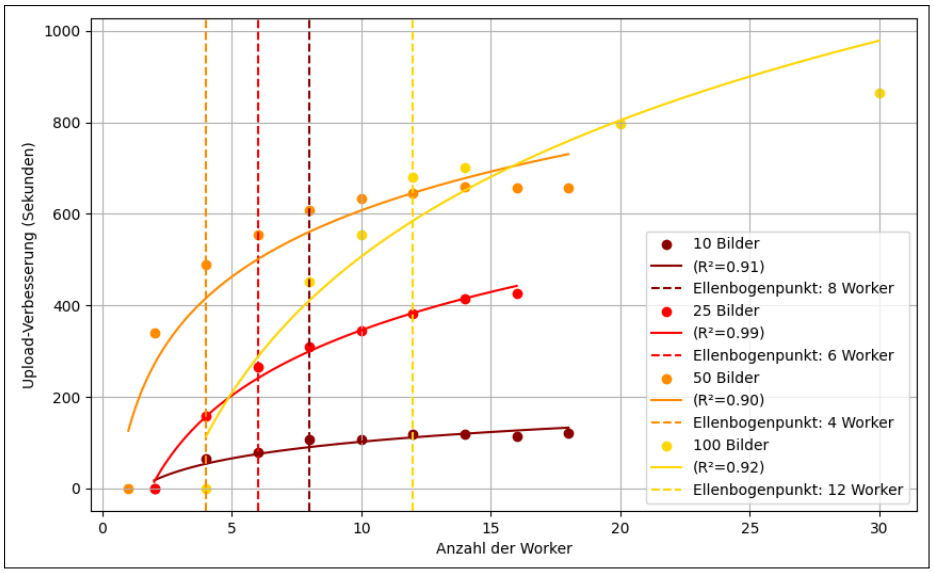

Abbildung 28: Verbesserung der Uploadzeit einer Zarr-Datei in Abhängigkeit der Anzahl der Worker

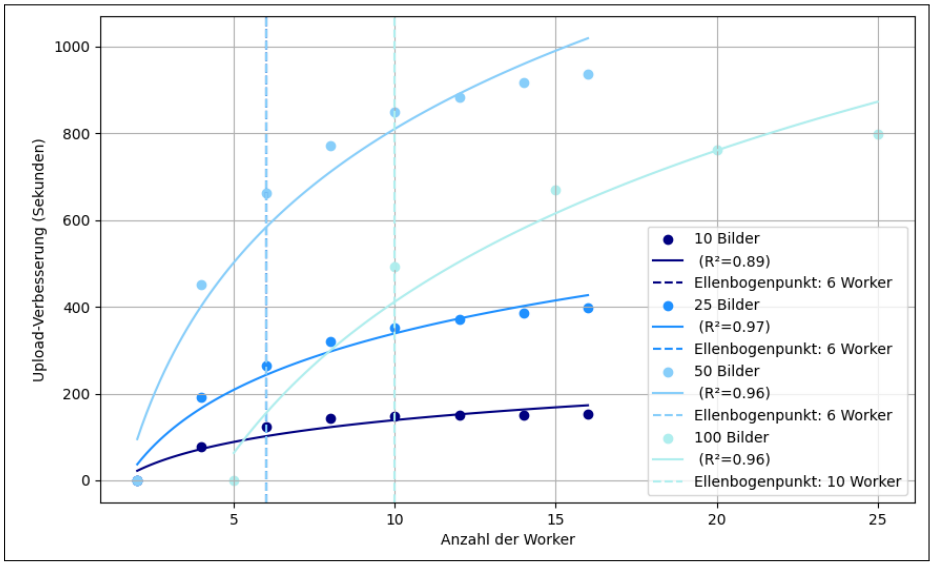

Abbildung 29: Verbesserung der Uploadzeit von COG-Dateien in Abhängigkeit der Anzahl der Worker

Insgesamt zeigen die Ergebnisse, dass große Datenmengen auch mit einer moderaten Anzahl von Workern effizient verarbeitet werden können, da der Verarbeitungsaufwand langsamer steigt als die Datenmenge. Zwar ist es sinnvoll, mit zunehmender Datenmenge mehr Worker einzusetzen, jedoch führt eine Erhöhung ab einem bestimmten Punkt zu keinen signifikanten Leistungsgewinnen mehr. Für die nachfolgenden Untersuchungen dieser Arbeit wird deshalb eine konstante Anzahl von zwölf Workern verwendet, um eine Balance zwischen Leistung und Ressourcenverbrauch sicherzustellen. Diese Entscheidung basiert auf der Analyse von 100 DOPs, bei der der Ellenbogenpunkt zwischen zehn und zwölf Workern liegt und führt auch bei kleineren Datenmengen nicht zu übermäßigem Overhead. Zusammenfassend wird auf Basis der durchgeführten Testmessungen folgende Justierung der Dask-Parameter für die Verarbeitung großer raumzeitlicher Rasterdatensätze in der CodeDE-Umgebung identifiziert und für alle Untersuchungsszenarien aus Kapitel 4.3 angewandt:

- Anzahl der Worker: 12
- Threads pro Worker: 2
- Arbeitsspeicher pro Worker: 10 GB

6.3 Speicherstrategien für raumzeitliche Rasterdaten in COG und Zarr

Wie in Kapitel 5 beschrieben können raumzeitliche Rasterdaten je nach Datenformat unterschiedlich organisiert werden, was die Effizienz von Abfragen und Analysen maßgeblich beeinflusst. In diesem Kapitel werden die technischen Umsetzungsmöglichkeiten für die Speicherung der DOPs mit COG und Zarr hinsichtlich ihrer Vor- und Nachteile für diese Untersuchung abgewogen.

Die Erstellung einer COG-Datei mit raumzeitlichen Rasterdaten kann technisch auf verschiedene Weise erfolgen. Bei der Entscheidung für eine Speicherung im Stil des Snapshot-Modells stellt jede COG-Datei einen räumlichen Ausschnitt zu einem bestimmten Zeitpunkt dar (siehe Abbildung 30 Option 1a und 1b). Die Zeitdimension wird dabei als Attribut in den Metadaten angegeben. Bei Option 1 muss weiterhin entschieden werden, wie groß die einzelnen räumlichen Ausschnitte sein sollen. Die Entscheidung, ob das gesamte Untersuchungsgebiet in einer Datei oder in kleineren Teilbereichen gespeichert wird, erfordert eine Abwägung zwischen Performance und Flexibilität. Kleinere Dateien reduzieren Speicherbedarf und ermöglichen gezieltere Verarbeitung und Visualisierung, sind jedoch weniger effizient bei großflächigen Abfragen, da mehr HTTP-Requests notwendig sind. Größere Dateien verringern den Verwaltungsaufwand und sind bei großflächigen Abfragen effizienter, benötigen jedoch mehr Speicherplatz und sind weniger flexibel. Änderungen erfordern stets die Neuerstellung der gesamten Datei, da COG keine Modifikationen erlaubt. In Option 2 würden Zeitpunkte als Bänder innerhalb einer Datei gespeichert, was den Verwaltungsaufwand reduziert, aber die Nutzung erschwert, wie in Kapitel 5 beschrieben. Während weniger HTTP-Requests notwendig sind, erschwert die Integration in gängige GIS-Tools die Analyse. Viele GIS lesen standardmäßig nur die ersten drei Bänder, was Anpassungen oder Mehrfachladen erfordert. Mit wachsender Bandanzahl steigt die Dateigröße, wodurch Software- und Hardwaregrenzen problematisch werden können. Änderungen oder Ergänzungen von Zeitpunkten erfordern ebenfalls die Neuerstellung der gesamten Datei.

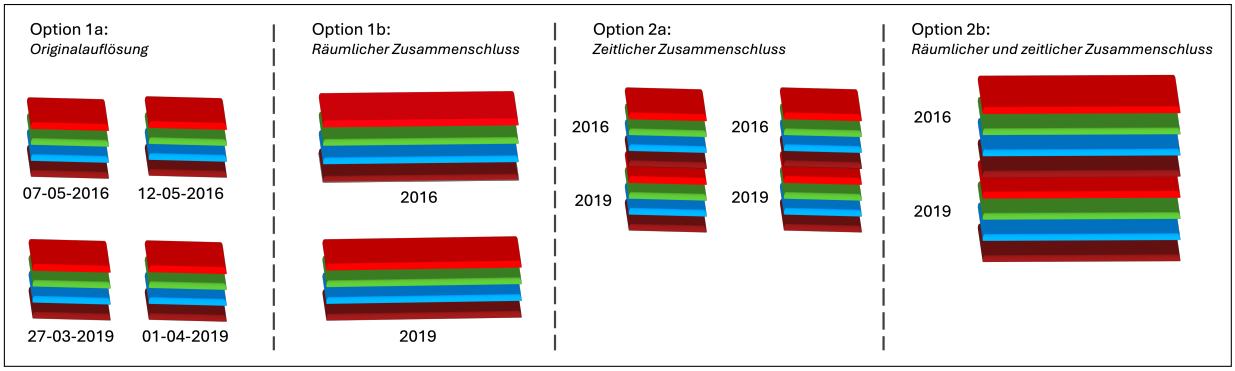

Abbildung 30: Technische Ansätze zur Speicherung raumzeitlicher Rasterdaten mit COG (Eigene Darstellung)

Option 2 ist in der Praxis weniger benutzerfreundlich und mit etablierten Workflows weniger kompatibel. Rasterdaten, wie sie etwa vom LGLN bereitgestellt werden, werden üblicherweise als kleine, räumlich aufgeteilte Einzeldateien organisiert. Diese Speicherform dient als Referenzpunkt, um ihre Eignung für die effiziente Verarbeitung und Analyse raumzeitlicher Rasterdaten im Vergleich zum Data Cube-Ansatz zu prüfen. In dieser Untersuchung wird das COG-Format nach Option 1a umgesetzt, wobei jeder Zeitpunkt in eine separate Datei ausgelagert wird. Dies ermöglicht eine direkte Vergleichbarkeit mit der bestehenden Organisation der DOPs im LGLN.

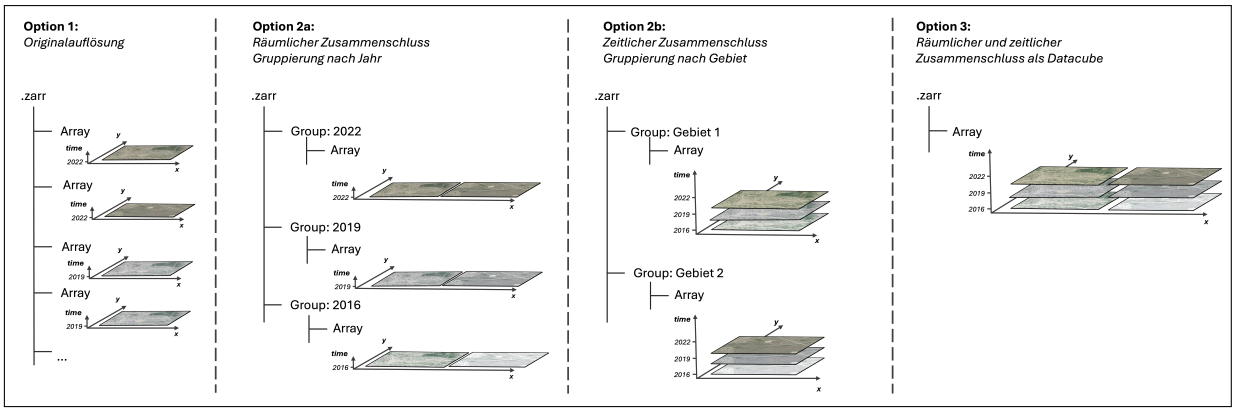

Abbildung 31: Technische Ansätze zur Speicherung raumzeitlicher Rasterdaten mit Zarr (Eigene Darstellung)

Auch die Erstellung der Zarr-Datei kann aufgrund der in Kapitel 5 beschriebenen hierarchische Katalogstruktur technisch unterschiedlich umgesetzt werden. Während Optionen 1 und 2 verschiedene Varianten der Gruppierung oder Trennung von DOPs bieten, nutzen sie die Stärken von Zarr wie parallelen Zugriff und effiziente Abfragen nicht vollständig aus (siehe Abbildung 31). In Option 1 werden DOPs einzeln gespeichert, was den Overhead erhöht. Option 2 gruppiert die Daten räumlich oder zeitlich, erleichtert jedoch nur spezifische Abfragen und bleibt ineffizient bei multidimensionalen Analysen. Partielle Zugriffe werden dabei nur bedingt unterstützt, da jedes Array seperat abgefragt werden muss. Option 3 hingegen bildet einen vollständigen Data Cube, in dem die Daten räumlich und zeitlich in einem einzigen Array integriert sind. Diese Struktur nutzt die Stärken von Zarr wie paralleles Lesen und Schreiben und chunkbasierte Modifizierung. Eine Unterteilung in Gruppen oder mehreren Arrays macht Sinn, wenn die gleichen Daten mit unterschiedlichen räumlichen oder zeitlichen Auflösungen vorliegen, unterschiedliche Metadaten wie Datenquellen oder Sensoren aufweisen oder wenn die Daten nur in Teilbereichen homogen vorliegen, während sie in der Gesamtheit große Lücken aufweisen. Auch eine thematische Trennung kann sinnvoll sein. Im Falle der DOPs vom LGLN bietet sich jedoch die Nutzung eines großen Arrays an, da die Daten für den Untersuchungsraum flächenhaft und homogen vorliegen.

6.4 Speicherung in der Cloud

Um die Untersuchung aus Kapitel 4.3.1 umzusetzen, wird nachfolgend ein Uploadprozess für COGs und Zarr entwickelt. Dieser Ansatz dient als Proof-of-Concept, um den Einfluss von Kachelgrößen und Kompressionsmethoden auf die Generierung der Dateien und den Upload in einer realitätsnahen Infrastruktur zu evaluieren. Dabei wird ein möglicher, effizienter und skalierbarer Ansatz untersucht, um zu prüfen wie gut die Formate eingesetzt werden können und welche Optimierungsmöglichkeiten sich ergeben. Die praktische Umsetzung erwies sich als zeitintensiv und technisch anspruchsvoll, was wertvolle Erkenntnisse für zukünftige Implementierungen lieferte. Im Folgenden wird der implementierte Code erläutert, einschließlich der gewählten Ansätze, der konkreten Umsetzung und der identifizierten Herausforderungen. Der grundsätzliche Ablauf erfolgt wie in Abbildung 17 dargestellt. Ziel des Codes ist es, hochauflösende DOPs mit verschiedenen Konfigurationen in das COG- und Zarr-Format zu konvertieren und anschließend effizient in einen Cloud-Bucket zu laden. Der vollständige Code, sowie die Auswertung ist zur Nachvollziehbarkeit im digitalen Anhang dieser Arbeit hinterlegt (Jupyter-Notebooks im Ordner Upload).

6.4.1 COG

Bei dem Upload von COG-Dateien werden neben Dask Bibliotheken wie Rioxarray, Rasterio, Pystac und Rio-cogeo eingesetzt, die speziell für die Geodatenverarbeitung optimiert sind. Obwohl die Dateigröße der DOPs beibehalten wird und die Daten bereits in Form von analysebereiten COG-Dateien vorliegen, können die DOPs vom LGLN nicht einfach in den eigenen Cloud-Bucket überführt werden, da die Daten

nicht über einen Zeitstempel verfügen, der für spätere raumzeitliche Abfragen essenziell ist. Darüber hinaus werden verschiedene Kachelgrößen und Kompressionsmethoden untersucht, die eine Neuerstellung der Datei notwendig machen. Die DOPs liegen unkomprimiert vor, sodass nur die relevanten Konfigurationen angepasst werden müssen, ohne aufwendige Vorverarbeitung.

Datenauswahl:
Zunächst dient die Funktion *searchItems* dazu, innerhalb einer STAC-API nach relevanten Datensätzen zu suchen (siehe Listing 4). Die Bibliothek Pystac wird dabei verwendet, um eine Verbindung zu der API mit der Kollektion „DOP“ unter der URL „https://dop.stac.lgln.niedersachsen.de“ herzustellen. Kriterien waren das Untersuchungsgebiet aus dem Kapitel 4.3, ein Zeitfenster und die maximale Anzahl an Datensätzen. Die Funktion liefert eine Liste von Items und deren Anzahl für die Verarbeitung zurück.

```
    def searchItems(area,max):
        api_url = "https://dop.stac.lgln.niedersachsen.de"
        client = STACClient.open(api_url)
        search_results = client.search(
            collections=["DOP"],
            intersects= area,
            datetime="2016-01-01/2024-01-01",
            max_items= max
        )
        items = search_results.item_collection()
        return items, len(items)
```

Listing 4: Datensuche

Vorbereitung:
Mit der Funktion *load_and_prepare* werden die Items geladen, das CRS festgelegt und der Zeitstempel hinzugefügt. Um die in Kapitel 6.3 festgelegte Datenstruktur umzusetzen, werden die Daten in ihrer Größe nicht verändert oder zusammengefügt, sondern einzeln vorbereitet und in den Cloud-Speicher hochgeladen. Hierzu werden die RGBI-Daten eines Items, die im Attribut *dop20_rgbi* als COG verlinkt sind, mithilfe von Rioxarray in Form eines Arrays geöffnet und angepasst:

```
rds = rioxarray.open\_rasterio(item.assets["dop20_rgbi"].href, chunks={'
    band': 1, 'x': chunk_size, 'y': chunk_size})
```

Listing 5: Datenvorbereitung

Die Kachelgröße wird dabei direkt angegeben, um späteres Rechunking zu vermeiden. Dies ermöglicht direkt den tile-basierten Zugriff auf die Datei. Zusätzlich wird aus den STAC-Metadaten des Items der Zeitstempel extrahiert und als Attribut (TIFFTAG_DATETIME) hinzugefügt. Die Metadaten und Attribute der Ausgangsdatei werden in diesem Workaround automatisch extrahiert und dem resultierenden Xarray.Array als Attribute angefügt. Die Daten werden als Xarray.Array zurückgegeben.

Die Funktion *process_and_upload* übernimmt die zweite und dritte Phase des Upload-Prozesses für das COG-Format. Dabei wird eine COG-Datei mit festgelegter Kachelgröße und Kompression aus einem Array erstellt und in einen Cloud-Bucket hochgeladen. COG-Dateien können nicht direkt in die Cloud geschrieben werden, da ihre spezielle interne Struktur erst vollständig erstellt werden muss. Damit der partielle Zugriff auf Teile der Daten funktioniert, müssen alle Tiles, IFDs und Metadaten vorab fertiggestellt und in einer bestimmten Reihenfolge in der Datei abgelegt werden. Deshalb muss vorab eine temporäre oder lokale Datei erstellt werden, die nach Fertigstellung in den Cloud-Speicher hochgeladen wird. Da der gesamte Prozess vollständig in der Cloud ausgeführt wird und der verfügbare Speicherplatz auf dem JupyterHub begrenzt ist, ist eine lokale Zwischenspeicherung aller Dateien nicht möglich. Um dies zu umgehen, wird ein Memory-File verwendet, das die Daten direkt im Arbeitsspeicher verarbeitet. Der Upload wird mit Boto3 realisiert. Boto3 übernimmt dabei die Authentifizierung mittels Access-Key und Secret-Key, das Verbindungsmanagement über die spezifische Endpoint-URL zum CloudFerro-Object-Bucket und die eigentliche Datenübertragung. Es bietet somit eine einfache Möglichkeit, Daten in den Cloud-Bucket hochzuladen, ohne sich um die technischen Details der Kommunikation kümmern

zu müssen. Boto3 wird verwendet, da es besser mit In-Memory-Dateien kompatibel ist als S3fs und in der Dokumentation empfohlen wird (cogeotiff, n. d., o.S.). Die Erstellung des Memory-Files ist im Anhang in Abschnitt A dargestellt.

Zur Erstellung einer COG-Datei wird ein COG-Profil mit der Bibliothek Rio-cogeo verwendet, einem Plugin für Rasterio zur Generierung und Validierung von COGs cogeotiff, n. d., o.S. Die Bibliothek bietet eine API mit vorkonfigurierten Profilen für verschiedene Kompressionsmethoden, die standardmäßig eine Kachelgröße von 512x512 Pixeln nutzen. Für die Untersuchung variabler Kachelgrößen und Kompressionen wird das Profil entsprechend der Konfigurationen aus Kapitel 4.3.1 angepasst (siehe Listing 6). Dabei wird für Szenarien ohne Kompression das „Raw"-Profil genutzt. Die Kompression wird über den Profilnamen definiert. Die Kachelgröße wird über die Parameter (*blockxsize* und *blockysize*) gesteuert. Zudem wird der *Alpha*-Parameter angepasst, um fehlerhafte Darstellungen der RGBI-DOPs zu vermeiden. Übersichten werden nicht generiert (*overview_level = 0*), da diese primär der Visualisierung dienen und für die Analyse nicht relevant sind. Ohne Übersichten ist ein besserer Vergleich mit Zarr-Dateien möglich, die nativ keine unterstützen. Zur Weiterleitung von Band-Metadaten wie Statistiken wird *forward_band_tags* auf True gesetzt.

```
    cog_profile = cog_profiles.get(com_typ)
    if chunk_size:
        cog_profile.update(blockxsize=chunk_size)
        cog_profile.update(blockysize=chunk_size)
    cog_profile.update(
        ALPHA="NO",
        overview_level = 0,
        forward_band_tags=True
    )
```

Listing 6: Aktualisierung des COG-Profils

Mithilfe der Methode *cog_translate* von Rio-cogeo wird das MemoryFile in das COG-Format umgewandelt. Als Parameter werden dabei das MemoryFile, der Name der Datei, das angepasste COG-Profil, sowie die Indizes der Bänder, die aus der Eingabe-Datei in das COG geschrieben werden sollen, übergeben. Mit dem Parameter *in_memory=True* wird die Verarbeitung komplett im Speicher ausgeführt, ohne auf das lokale Dateisystem zu schreiben (siehe Anhang in Abschnitt A Funktion process_and_upload).

Upload:
Das erzeugte COG wird mit einem eindeutigen Dateinamen über den Boto3-Client direkt aus dem Speicher per *upload_fileobj* in den Bucket hochgeladen. Der Dateiname setzt sich aus dem Zähler *count* (Anzahl der DOPs), dem Array-Namen (*rds.name*) und der verwendeten Chunk-Größe oder Kompressionsmethode zusammen:

```
    s3_key = f"{count}_{rds.name}_partition_{chunk_size}.tif"
    s3.upload_fileobj(cog_memfile, bucket_name, s3_key)
```

Listing 7: Upload der COG-Datei

Ausführung:
Die Ausführung der Untersuchungen erfolgt in einer Schleife mit den verschiedenen Kachelgrößen oder Kompressionsmethoden. Vorab wird, wie in Kapitel 6.1 beschrieben, mit der Methode *createGateway-Cluster* ein Dask-Cluster mit den in Kapitel 6.2 festgelegten Parametern erstellt. Dieses Cluster ermöglicht die parallele Verarbeitung und wird für jeden Test neu initialisiert, um potenzielle Speicherlecks durch unmanaged Memory zu vermeiden. Die Performance wird durch die in Kapitel 2.4.3 beschriebenen Performance-Reports und einen Memory-Sampler überwacht. Die Zeitmessung erfolgt für das Vorbereiten und den Upload der Daten jeweils separat, während der Memory-Sampler die Speichernutzung für alle Testläufe in einem Diagramm darstellt. Für die Untersuchung der verschiedenen Kompressionsmethoden wird das COG-Profil um einen zusätzlichen Parameter für den Prädiktor ergänzt (*PREDICTOR=2*), um Testsfälle mit dem Prädiktor 2 durchführen zu können.

In der Durchführungsschleife werden die Datensätze mit der Funktion *searchItems* basierend auf geografischen Kriterien gesucht, geladen und mit den Funktionen *load_and_prepare* sowie *process_and_upload* verarbeitet und in den CODE-DE-Cloud-Bucket geladen. Dabei werden die Kachelgrößen und Kompressionsmethoden als Parameter übergeben. Zur Implementierung wird *dask.delayed* genutzt, um die Funktionen *load_and_prepare* und *process_and_upload* für jedes zu verarbeitende Item zu verzögern. *Dask.delayed* ermöglicht die in Kapitel 2.4.1 erwähnte Lazy Evaluation, bei der Funktionen nicht sofort ausgeführt. Jeder Aufruf von *dask.delayed* erzeugt ein verzögertes Objekt, das lediglich die Metadaten und den Funktionsaufruf beschreibt, ohne die Funktion direkt auszuführen. Nachdem alle Aufgaben gesammelt sind, erstellt Dask einen Task-Graphen, der die Abhängigkeiten und Reihenfolge der Aufgaben definiert. Der Graph zum Upload einer COG-Datei wird ausschnittsweise in Abbildung 32 dargestellt. In diesem Beispiel betrug die Kachelgröße 10.000 x 10.000 Pixel, sodass die drei Dateien bei vier Spektralbändern jeweils in vier Chunks geladen werden (open_rasterio). Diese Chunks werden anschließend mit einem *finalize* von Dask zusammengeführt, da COG das kachelbasierte Schreiben nicht unterstützt. Mit dem Ergebnis wird dann die Methode *process_and_upload* aufgerufen, um aus dem Array eine COG-Datei zu erstellen und diese in den Cloud-Bucket zu laden. Der Task-Graph wird anschließend vom Dask-Scheduler verwendet, um die Aufgaben auf verfügbare Worker zu verteilen und deren Auslastung zu optimieren. Das Sammeln aller Aufgaben und die Erstellung eines zentralen Task-Graphen ermöglicht eine effizientere Ausführung, da der Scheduler die Ressourcen besser auf die Worker verteilen kann, im Vergleich zu einer sequentiellen Aufgabenübermittlung. Erst bei der Ausführung von *dask.compute* werden die Aufgaben parallel abgearbeitet.

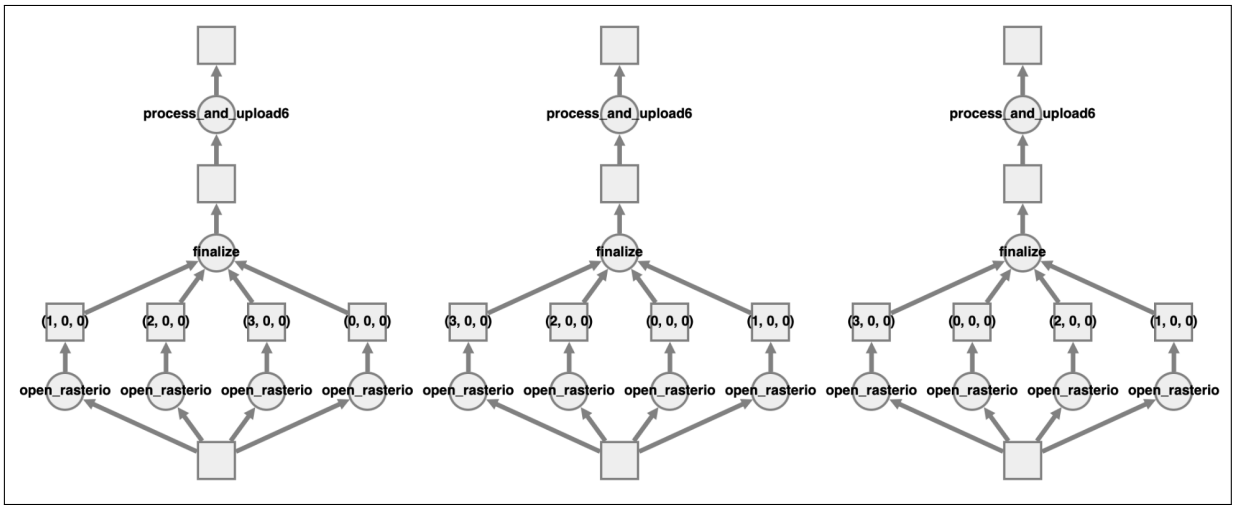

Abbildung 32: Ausschnitt des Task-Graphen für den Upload von COG-Dateien

Ein Nachteil von *dask.delayed* ist die Skalierbarkeit bei sehr großen Workflows. Mit zunehmender Aufgabenanzahl kann der Task-Graph die Performance des Schedulers beeinträchtigen und und das in Kapitel 6.2 beschriebene Problem des Task-Overheads hervorrufen. Tabelle 7 zeigt, dass bei kleinen Kachelgrößen die Anzahl der zu verarbeitenden Dask-Chunks stark steigt. Bei einer Kachelgröße von 256 × 256 Pixeln für 100 DOPs müssen bis zu 640.000 Chunks verarbeitet werden. Zudem fällt auf, dass die Dask-Chunkgrößen sehr klein sind. Laut den Dask-Best-Practices ist es ratsam Größen von mindestens 100 MB zu verwenden (Buckley, 2021, o.S.). Die Verarbeitung von Chunk-Größen unter ein MB ist fast immer unzureichend. Die Standard-Kachelgröße eines COG beträgt jedoch 512 x 512 Pixel was bei den hochauflösenden DOPs lediglich 0,2 MB bedeutet. Diese Kombination führte zu Instabilitäten wie Netzwerkproblemen und Überlastungsengpässen, da der Scheduler überwiegend mit der Verteilung neuer Aufgaben beschäftigt war.

Kachel-größe	Chunk-größe [MB]	Chunk-Anzahl 1 DOP	Chunk-Anzahl 9 DOPs	Chunk-Anzahl 54 DOPs	Chunk-Anzahl 100 DOPs
256	0,07	6.400	57.600	345.600	640.000
400	0,16	2.500	22.500	135.000	250.000
512	0,26	1.600	14.400	84.400	160.000
1.024	1,05	400	3600	21.600	40.000
2.000	4,00	100	900	5.400	10.000
4.096	16,78	36	324	1.944	3.600
8.192	67,11	16	144	864	1.600
10.000	100,00	4	36	216	400

Tabelle 7: Anzahl und Größe der Chunks von Dask-Arrays in Abhängigkeit der Kachelgröße

Alternativen wie die Teilung in kleinere Batches oder die Nutzung der Futures-API wurden in Erwägung gezogen, führten jedoch nicht zu einer stabilen Durchführung. Selbst bei Batch-Größen von maximal 10 DOPs traten neben massiven Leistungseinbußen weiterhin Instabilitäten auf. Auch der Einsatz anderer Bibliotheken wie *odc.geo.cog.save_cog_with_dask* brachte keine Lösung. Die begrenzte Kontrolle über den Task-Graphen und die Herausforderungen beim Debugging in verteilten Umgebungen erschwerten die Optimierung zusätzlich. Trotz intensiver Bemühungen, eine stabilere Leistung zu erzielen, zeigte sich daher, dass die Kombination aus der Datenstruktur der hochauflösenden DOPs des LGLN, Dask und dem COG-Format für diesen Anwendungsfall nicht optimal ist.

Die einzige praktikable Lösung bestand darin, die Kachelgrößen zu vergrößern, um dem Scheduler größere und besser planbare Aufgaben zu übergeben. Dies hatte jedoch zur Folge, dass einige Testszenarien nicht realsierbar waren. Die Testfälle mit Kachelgrößen von 256 × 256 und 400 × 400 konnten aufgrund der beschriebenen Instabilitäten nicht in die Analyse einbezogen werden. Die verbleibenden Tests werden standardmäßig dreimal wiederholt, um die Konsistenz der Ergebnisse zu gewährleisten. In einigen Fällen waren aufgrund von Instabilitäten zusätzliche Wiederholungen erforderlich. Nach jedem Durchlauf wird der erfolgreiche Abschluss des Uploads überprüft und die Ressourcennutzung mittels Performance-Report und MemorySampler überwacht. Bei Problemen wird der Test neu gestartet.

6.4.2 Zarr

Der Upload von Zarr-Dateien mit unterschiedlichen Kachelgrößen und Kompressionsmethoden wird mit Xarray, S3fs und Dask umgesetzt. Die Datensuche über STAC erfolgt dabei analog zu Kapitel 6.4.1 mit der Funktion *searchItems*.

Vorbereitung:
Im Gegensatz zu COG ist die Datenvorbereitung etwas umfangreicher, da neben der Vorbereitung der einzelnen Arrays auch eine räumliche und zeitliche Zusammenführung vorgenommen wird, um einen Data Cube zu erstellen. Die Vorbereitung der einzelnen Arrays gliedert sich dabei in drei Schritte, der zeitlichen Harmonisierung, der Festlegung des Koordinatenreferenzsystems und der Optimierung der Datentypen:

1. **Zeitliche Harmonisierung:** Analog zur Generierung von COG-Dateien werden die ausgewählten STAC-Items mit Rioxarray in Form von Arrays geladen. Statt vollständiger Zeitstempel wird jedoch nur das Aufnahmejahr berücksichtigt, da die DOPs in Niedersachsen kleinräumlich uneinheitliche Aufnahmetage vorweisen. Dies würde zu Inkonsistenzen und einer großen Menge Nullwerte in dem Data Cube führen. Da im Untersuchungsraum innerhalb eines Jahres die Daten nicht mehrfach vorliegen, kann eine konsistente Zeitachse erzeugt werden, indem nur das Aufnahmejahr als Zeitdimension hinzugefügt wird.

2. **Festelgung des CRS:** Die korrekte Speicherung des CRS für Zarr-Dateien erfordert einen Workaround, der aufgrund der fehlenden Dokumentation von Zarr Experimentieren erfordert. Rioxarray setzt zwar das CRS korrekt, dies wird jedoch nicht direkt von Zarr erkannt. Um das CRS konsistent zu verankern, wird es sowohl in den Koordinaten als auch in den Attributen definiert, damit es in Xarray sichtbar ist und für alle Arrays innerhalb des Datensatzes gilt. Für die Erstellung einer Zarr-Datei ist es entscheidend, die Attribute *Area_of_Point* und *_CRS* präzise und in der erforderlichen Schreibweise zu setzen. Das CRS wird im Attribut *_CRS* in Form eines JSON-Objekts abgelegt, welches die vollständige WKT-Beschreibung des Koordinatenreferenzsystems enthält. Die Angabe des EPSG-Codes allein ist hierbei nicht ausreichend. Dies garantiert die Kompatibilität mit GIS-Systemen und Geodatenbibliotheken.

3. **Festlegung der Datentypen:** Ein weiterer wichtiger Aspekt ist die Festlegung des Datentyps. Um Speicherplatz zu sparen und Fehler zu vermeiden, wird der Datentyp des Xarray-Arrays explizit auf uint8 geändert, da die RGBI-Daten Werte nur von 0 bis 255 enthalten. Ohne diese Anpassung würde der Standard-Datentyp float64 von *Xarray* angenommen. Dieser benötigt nicht nur erheblich mehr Speicherplatz, sondern ist für RGBI-Daten auch ungeeignet und kann bei der Geodatenverarbeitung wie z. B. Koordinatentransformationen zu Fehlern führen. Darüber hinaus übernimmt Zarr automatisch den Datentyp des übergebenen Arrays. Dieses Problem tritt speziell bei mehrdimensionalen Arrays auf und war bei der Generierung von COG-Dateien nicht relevant.

Nach der Umwandlung jedes STAC-Items in ein mehrdimensionales Array, werden die Arrays im zweiten Schritt mithilfe der Methode *merge_datasets_by_time* zusammengeführt. Die Zusammenführung erfolgt mit der Methode *combine_by_coords*. Diese Methode bietet eine performante und speichereffiziente Zusammenführung basierend auf standardisierten Koordinaten und ermöglicht es, die Ausführung lazy vorzunehmen. Die Methode entspricht damit den Dask-Best-Practices, indem Berechnungen erst am Ende ausgeführt werden, um Parallelisierung zu optimieren. Alternativen wie *concat* oder *merge* werden aufgrund von Problemen mit redundanten Zeitpunkten, Ressourcenengpässen und ineffizienter Verarbeitung verworfen. Für die Nutzung von *combine_by_coords* war es notwendig, den Datentyp der Nullwerte explizit auf uint8 festzulegen, um Kompatibilitätsprobleme zwischen den vorliegenden Integer-Werten und dem Standarddatentyp float64 zu vermeiden. Zudem musste ein Umgang mit den Attributen der Arrays gefunden werden, die spezifische Informationen aus den ursprünglichen COG-Dateien des LGLN enthalten wie Pixelrepräsentation und Bandstatistiken. Die Funktion bietet für den Umgang mit abweichenden Attributen die Möglichkeit, alle Attribute zu entfernen, die Attribute jeweils zu überschreiben oder nur gemeinsame Attribute zu behalten. Um möglichst viele dieser Attribute zu erhalten, werden nur identische Werte beibehalten. Diese Vorgehensweise gewährleistet eine konsistente Zusammenführung der Arrays ohne Datenkonflikte:

```
1
2     combi_dops = xr.combine_by_coords(datasets, combine_attrs="
    drop_conflicts", fill_value=np.uint8(0))
```

Listing 8: Zusammenführung mehrerer mehrdimensionaler Arrays

Abschließend musste erneut der Datentyp der Arrays von float64 auf uint8 geändert werden. Abbildung 33 zeigt das zusammengeführte, mehrdimensionale Array in *Xarray*. Es umfasst vier Zeitstufen (2013–2022), vier Bänder und eine räumliche Ausdehnung von 50.000 x 50.000 Pixeln, gespeichert als uint8 in 400 Dask-Chunks (jeweils 10.000 x 10.000 Pixel) mit einem Speicherbedarf von 37,25 GiB. Zusätzlich enthält es eine Referenz für das Koordinatensystem (spatial_ref). Die gesetzten Attribute befinden sich auf der Ebene der Datenvariable, da globale Dataset-Attribute im GIS nicht dargestellt werden können.

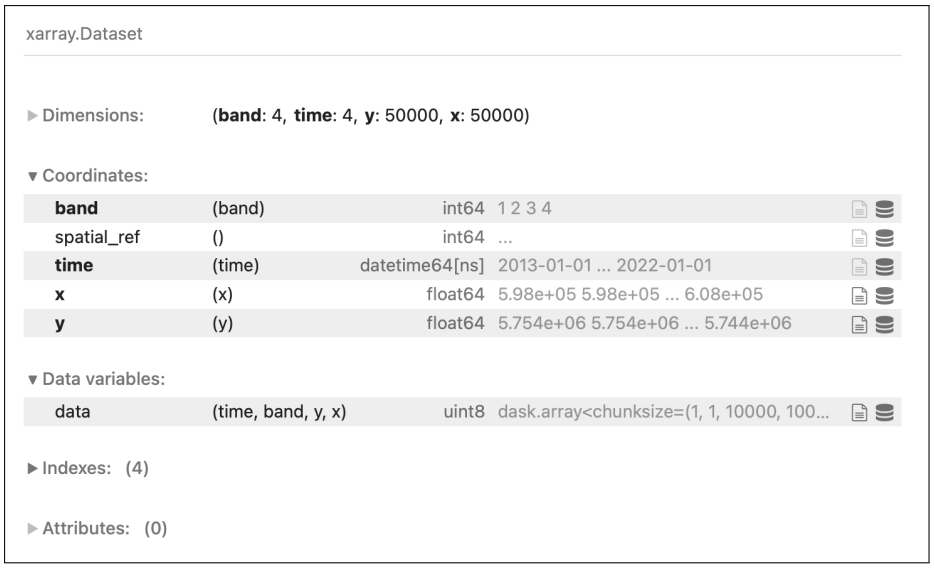

Abbildung 33: Darstellung einer Zarr-Datei in Xarray

Upload:
Die Funktion *uploadToBucket* ermöglicht den Upload des Arrays in Form einer Zarr-Datei auf einen S3-kompatiblen Cloudspeicher. Zunächst werden die Zugriffsparameter (Access- und Secret-Key) sowie die Verbindung zum Object-Bucket über die Bibliothek S3fs eingerichtet. Die Verbindung erfolgt analog zu Boto3 mithilfe des definierten Endpunkt-URLs zur CODE-DE-Cloud. Für die Untersuchungsfälle mit verschiedenen Kompressionsmethoden wird der entsprechende Algorithmus über zarr.storage.default_compressor aktiviert, andernfalls wird die Komprimierung deaktiviert. Ohne diesen Parameter werden die Daten standardmäßig mit dem Blosc-Kompressor und dem LZ4-Algorithmus komprimiert. Anschließend wird der Pfad der Zarr-Datei im Bucket definiert. Die Daten werden mithilfe von s3fs.S3Map einem Speicherort zugeordnet, sodass die Zarr-Datei direkt dorthin geschrieben werden kann. Für den Upload wird eine spezifische encoding-Konfiguration definiert. Diese legt unter anderem den *_FillValue* (Nullwert) für fehlende Datenpunkte fest, definiert die Chunks entsprechend der angegebenen Kachelgröße und stellt sicher, dass der Datentyp uint8 beibehalten wird. Schließlich erfolgt der eigentliche Upload über die Methode *to_zarr*, die das Xarray-Objekt direkt als Zarr-Datei speichert:

```
    encoding = {
        'data': {
            '_FillValue': np.uint8(0),
            'chunks': (1,1,chunk_size,chunk_size),
            'dtype': 'uint8'
        }
    }
    rds.to_zarr(store=s3map, consolidated=True, encoding= encoding, mode='w
    ', write_empty_chunks=False)
```

Listing 9: Erstellung und Upload einer Zarr-Datei

Der Parameter *consolidated=True* sorgt für die Konsolidierung der Metadaten. Der Modus *w* sorgt dafür, dass bestehende Dateien überschrieben werden und mit der Option *write_empty_chunks=False* werden leere Chunks vermieden, um den Speicherplatz weiter zu optimieren. Sofern die Dateien mit Xarray oder Zarr (zarr.open) anschließend geöffnet werden, enthalten sie allderdings wieder die Nullwerte und haben einen höheren Speicherbedarf. Das bedeutet, dass insbesondere bei der Verarbeitung beachtet werden muss, das die zu verarbeitende Datenmenge unter Umständen sehr viel größer ist, als die tatsächlichen Daten.

Durchführung:
Die Durchführung des Uploadprozesses erfolgt analog zu Kapitel 6.4.1 in einer Schleife mit den verschiedenen Kachelgrößen oder Kompressionsmethoden. Dabei muss zur Nutzung verschiedener Kompressoren die Bibliothek *Numcodecs* und *Zarr* als Plugin im Cluster nachinstalliert werden. Auch beim Upload von Zarr wird ein Performance-Report für jede Funktion sowie ein Plot für die Speichernutzung mit dem MemorySampler erstellt. In der Durchführung der Untersuchung wird zunächst die Funktion *searchItems* verwendet, um die zu verarbeitenden Datensätze zu suchen. Jedes DOP-Item wird durch die Funktion *load_and_prepare_rds* mit der entsprechenden Kachelgröße geladen und vorbereitet. Analog zur Durchführung mit COG-Dateien wird dies über *dask.delayed* gelöst. Dabei werde die COG-Dateien kachelweise geöffnet und um eine zusätzliche Zeitdimension erweitert. Danach besteht die Chunk-Bezeichnung aus vier Elementen anstelle von drei (z. B. 0,3,0,0). Im Gegensatz zu COG-Dateien, bei denen für das Schreiben ein spezieller Workaround über *dask.delayed* oder andere APIs notwendig ist, nutzen Zarr und Xarray nativ Dask für die parallele Verarbeitung. Dies macht die Implementierung deutlich einfacher und benutzerfreundlicher, da Funktionen wie *merge_datasets_by_time* automatisch lazy ausgeführt werden, ohne dass zusätzliche APIs oder manuelles Hinzufügen zu einem Task-Graph erforderlich sind. Erst beim Upload, bzw. beim Aufruf von *rds.to_zarr*, wird der komplette Code ausgeführt. Außerdem wird dabei direkt die chunkbasierte Struktur verwendet, um die einzelnen Chunks unabhängig und parallel in die Cloud hochzuladen. In Abbildung 34 wird deutlich, dass Zarr paralleles Schreiben unterstützt. Jede COG-Datei (Viereck ganz links) wird in vier Chunks aufgeteilt, wie durch den Stapel aus vier Vierecken im Task-Graph erkennbar ist. Diese vier Chunks pro Datei bleiben während des gesamten Prozesses erhalten, werden unabhängig voneinander verarbeitet und direkt in die Zarr-Datei geschrieben. Im Gegensatz dazu werden bei COG die vier Chunks nach der Verarbeitung durch *load_and_prepare* wieder zu einem einzigen Datensatz zusammengeführt, der erst danach hochgeladen wird. Die Farben im Task-Graph zeigen den Verarbeitungsstatus: Blau für abgeschlossene, Rot für wartende und Grün für aktive Tasks.

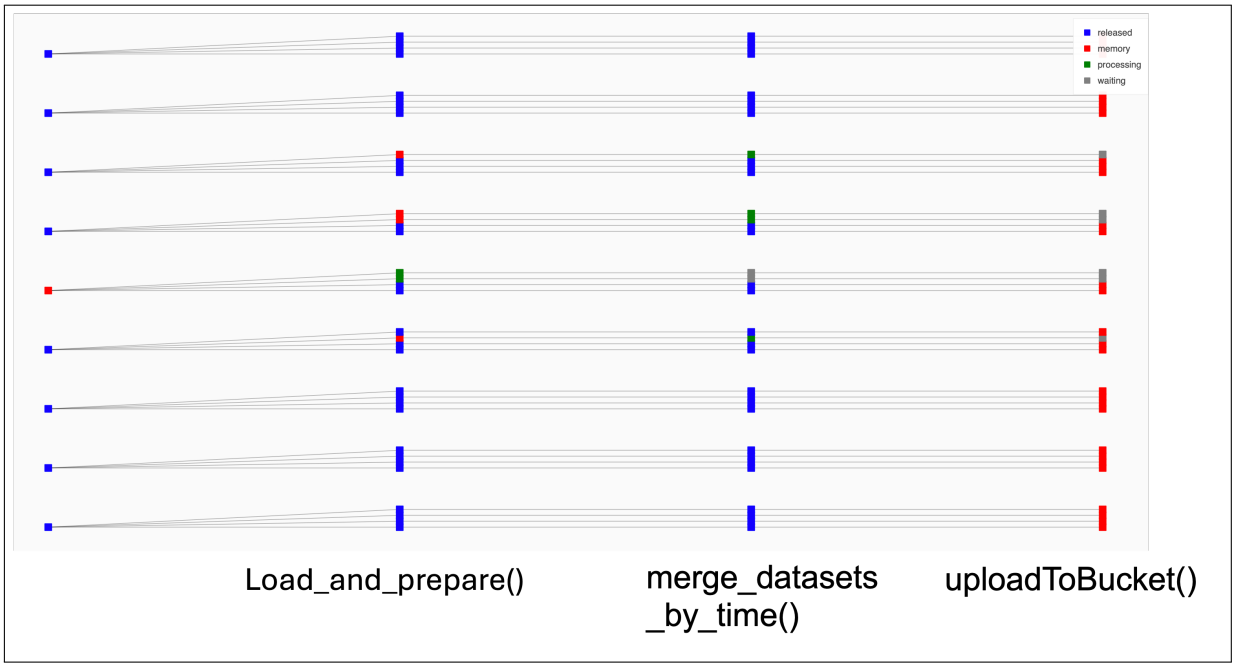

Abbildung 34: Ausschnitt des Task-Graphen im Dask-Dashboard bei Erstellung einer Zarr-Datei

Aufgrund von Instabilitäten, wie sie bereits in Kapitel 6.4.1 beschrieben wurden, konnten Tests mit kleinen Kachelgrößen (256 × 256, 400 × 400 und 512 × 512) bei mittleren und großen Datenmengen nicht erfolgreich durchgeführt werden. Diese Tests führten zu wiederholten Fehlern und werden aus der Analyse ausgeschlossen. Für die verbleibenden Szenarien wird jeder Test mindestens dreimal wiederholt, wobei instabile Durchläufe häufiger neu gestartet werden mussten, um konsistente Ergebnisse zu gewährleisten.

6.5 Implementierung raumzeitlicher Analysen

Zur Umsetzung der in Kapitel 4.3.2 beschriebenen Untersuchung werden Analyseprozesse für COGs und Zarr entwickelt. Dieses Kapitel beschreibt die Implementierung eines skalierbaren Ansatzes zur Evaluation beider Formate, mit dem Ziel, Effizienz und Handhabbarkeit im Kontext raumzeitlicher Analysen zu bewerten. Die Untersuchung gliedert sich in zwei Teile. Mit einer Analyse der Zugriffs- und Lade-Performance wird untersucht, welche Chunking- und Kompressionseinstellungen den effizientesten Zugriff ermöglichen. Zudem wird die Datenstruktur von ARD und Data Cubes im Hinblick auf raumzeitliche Abfragen evaluiert. Der zweite Abschnitt widmet sich der Umsetzung des in Kapitel 4.3.2 beschriebenen Anwendungsszenarios, das die Berechnung und Analyse des NDVI zur Erfassung von Vegetationsveränderungen über die Zeit umfasst. Aufgrund der zeitintensiven Implementierung war es nicht möglich, die Analyse vollständig abzuschließen und konkrete, vergleichbare Messwerte zu erzielen. Jedoch liefert die Erläuterung des implementierten Codes wertvolle Einsichten zu den Ansätzen, deren Umsetzung und den identifizierten Herausforderungen. Der vollständige Code ist im digitalen Anhang dieser Arbeit hinterlegt (Jupyter-Notebooks im Ordner Analyse). Erste Schlussfolgerungen aus den gewonnenen Erkenntnissen bilden eine Grundlage für weiterführende, detaillierte Analysen aufbauend auf dieser Arbeit. Aus den gewonnenen Erkenntnissen werden erste Schlussfolgerungen für die Evaluation der beiden Formate hinsichtlich der Nutzung und möglicher Optimierungspotenziale abgeleitet.

6.5.1 Raumzeitliche Abfragen

Ziel dieses Abschnittes ist es, die Performance von COG-Dateien bei verschiedenen raumzeitlichen Abfragen systematisch zu evaluieren. Dafür werden Metriken wie HTTP-Requests, übertragene Datenmengen, Anzahl der Tasks und die Abfragedauer unter verschiedenen Chunk- und Kompressionseinstellungen sowie für unterschiedliche räumliche, zeitliche und raumzeitliche Abfragebereiche analysiert. Ziel ist es, die Effizienz von Zarr als Speicherformat im Vergleich zu COG zu bewerten. Der Code kombiniert mehrere leistungsstarke Python-Bibliotheken, darunter S3fs, Xarray, Rioxarray und Dask.

Für die Analyse wird analog zum bisherigen Vorgehen ein Dask-Cluster eingerichtet. Zur Überwachung von HTTP-Requests und übertragenen Datenmengen wird ein spezialisiertes Filesystem *CountingS3FileSystem* entwickelt, das HTTP-Anfragen und deren Datenvolumen analysiert. Dieses erweitert die *s3fs.S3FileSystem*-Bibliothek, um HTTP-Requests und die damit verbundenen Datenmengen zu zählen. Beim Laden von COG-Dateien mit Rioxarray und S3fs zeigte sich jedoch, dass Teilbereiche nicht partiell geladen, sondern die gesamte Datei heruntergeladen wird. Dies widerspricht dem Ziel effizienter, skalierbarer Abfragen und machte diesen Ansatz unbrauchbar. Als Alternative wird ein Ansatz entwickelt, der auf S3fs verzichtet und stattdessen GDAL direkt nutzt. Dabei werden die erforderlichen AWS-Credentials (Access Key, Secret Key, Region) sowie spezifische Einstellungen wie das Deaktivieren von Virtual Hosting (*AWS_VIRTUAL_HOSTING = False*) gesetzt, um auf den Bucket in der CODE-DE-Cloud zugreifen zu können. Anschließend erfolgt der Zugriff über das */vsis3/*-Format von GDAL, das partielles Lesen ermöglicht.
Dabei wird die Methode *rioxarray.open_rasterio* genutzt, um auf räumliche Teilbereiche der Dateien zuzugreifen. Anders als bei der Bibliothek Rasterio wird kein Window und keine Bounding-Box zum partiellen Zugriff definiert. Bei Rioxarray erfolgt der partielle Zugriff über Slicing. Dabei werden die Koordinaten der Bounding-Box für die x- und y-Dimension als Slice in einem *.sel*-Argument angegeben (siehe Listing 10).

```
raster = rioxarray.open_rasterio(file_path).sel(x=slice(minx, maxx), y=slice(maxy, miny))
```

Listing 10: Öffnen einer COG-Datei mit Rioxarray

Zusätzlich werden die Zeitstempel aus dem Attribut *TIFFTAG_DATETIME* der Dateien extrahiert, um ein mehrdimensionales Array zu erzeugen. Für eine Vergleichbarkeit mit der Analyse der Zarr-Dateien wird dabei lediglich das Aufnahmejahr als zusätzliche Dimension integriert.

Durch den Verzicht auf S3fs können keine HTTP-Requests und Datenmengen gezählt werden. Alternativ können in einer lokalen Umgebung GDAL-Logs über *CPL_DEBUG=ON* analysiert werden, was in der Cloud jedoch nicht praktikabel war. Aufgrund der Lazy-Ausführung von Dask sowie der fehlenden GDAL-Installation in der Cloud-Umgebung konnten keine HTTP-Requests in der Konsole verfolgt werden. Aus Effizienzgründen wird dabei ein Workaround gewählt, der analog zum Vorgehen aus Kapitel 6.4.2 zur Vorbereitung der Zarr-Datei ist. Dabei werden die räumlichen Ausschnitte mit dask.delayed geladen und anschließend über die Funktion *combine_by_coords* zu einem mehrdimensionalen Array zusammengefügt. Mit dem *compute()*-Befehl wird der angefragte Ausschnitt anschließend vollständig in den Arbeitsspeicher geladen, um die Berechnung mit Dask auszuführen. Durch die Nutzung von Dask, kann dabei mittels Performance-Report und dem MemorySampler die Dauer und die Anzahl der Tasks extrahiert werden. Die Ergebnisse der verschiedenen Konfigurationen und Testfälle werden systematisch in einer Tabelle dokumentiert. Eine Messung der HTTP-Requests für raumzeitliche Abfragen war in dieser Untersuchung daher nicht möglich. Die Implementierung ist im Anhang (B) dokumentiert. Erste Tests zeigten, dass der Zugriff ohne S3fs effizienter ist, was darauf hindeutet, dass der Zugriff partiell erfolgt, jedoch bleibt das Zusammenfügen raumzeitlicher Teilbereiche im Vergleich zu Zarr deutlich langsamer.

Bei Zarr-Dateien kann über den S3fs-Client mit *xarray.open_zarr* direkt auf die Daten zugegriffen werden. Dank der direkten Integration mit Dask erfolgt die Abfrage in diesem Schritt lazy, sodass zunächst nur Metadaten geladen werden. Dies erlaubt es, Teilbereiche des Datasets, zu definieren. Wie bei COG wird der raumzeitliche Abfragebereich über Slices bestimmt. Die gewünschten Daten werden durch die Methode *.sel()* ausgewählt und mit dem *compute()*-Befehl vollständig in den Arbeitsspeicher geladen (siehe Listing 11). Die Kombination aus Zarr und Dask ermöglicht den parallelen Abruf der benötigten Chunks, was eine effiziente Abfrage ermöglicht.

```
1     ds = xr.open_zarr(store=s3map, mask_and_scale=False)
2     data_subset = ds['data'].sel(
3                 x=slice(minx, maxx),
4                 y=slice(maxy, miny),
5                 time=slice(time_point1, time_point2)
6             ).compute()
```

Listing 11: Raumzeitliche Abfragen für Zarr-Dateien mit Dask

Für große Abfragebereiche, wie in den Testszenarien 5 und 6 aus Kapitel 4.3.2, führte die Nutzung von *compute()* bei der Ausführung zu Speicherproblemen, da der Dask-Scheduler aufgrund begrenzter Speicherressourcen überlastet wird. Dies liegt daran, dass im Helm-Chart für den Dask-Scheduler keine explizite Obergrenze für den Arbeitsspeicher definiert war, sodass der Standardwert von Dask von lediglich zwei GB verwendet wird. Im zugrundeliegenden Workflow (siehe Sequenzdiagramm in Kapitel 4.5) übermittelt der Client die raumzeitliche Abfrage an den Dask-Scheduler, der die Tasks auf verschiedene Worker verteilt. Sobald der Scheduler die Ergebnisse der Worker erhält, leitet er diese an den Client weiter. Wird dabei das Speicherlimit des Schedulers überschritten, kommt es zu einem Absturz des Clusters und der Client verliert die Verbindung. Die maximale Datenmenge, die auf diese Weise verarbeitet werden kann, hängt daher direkt von den verfügbaren Ressourcen des Dask-Schedulers ab.

Eine mögliche Lösung wäre die Erhöhung des Speicherlimits im Helm-Chart. Beispielsweise kann der Scheduler auf einem dedizierten Kubernetes-Knoten mit einem Speicherlimit von bis zu 128 GB ausgeführt werden. Ein dedizierter Scheduler-Knoten verhindert, dass andere Pods auf demselben Knoten betrieben werden, was die Nutzung für Mehrbenutzerumgebungen einschränkt. Selbst bei maximaler Auslastung eines Knotens bleibt die Verarbeitung großer Datenmengen ineffizient, da die benötigten Ressourcen proportional zur Datenmenge wachsen. In der aktuellen Implementierung wird das Limit auf 10 GB erhöht, was als Kompromiss angesehen wird, um kleinere Abfragen durchführen zu können, aber weiterhin andere Pods auf demselben Knoten zu betreiben und so einen Mehrbenutzerbetrieb zu ermöglichen. Die Anpassung des Limits im Helm-Chart erfolgt wie folgt:

```
    backend:
    imagePullSecrets:
      - name: "all-icr-io"
    scheduler:
      memory:
        limit: 10G
```

Listing 12: Konfiguration des Dask-Schedulers im Helm-Chart

Statt den gesamten Abfragebereich zu berechnen, kann Dask durch die Ausgabe von Stichproben entlastet werden. Dabei würden jedoch aufgrund der Lazy-Evaluation nur die Werte der Stichprobe berechnet, nicht der vollständige Abfragebereich. Eine andere Möglichkeit ist das direkte Zurückschreiben der berechneten Werte in eine Datei oder einen Cloud-Bucket, um Speicherplatz zu sparen. Dies verfälscht jedoch die Performancemessung durch den zusätzlichen Uploadprozess und löst nicht das Problem, dass für das Laden der Daten weiterhin ausreichend Ressourcen erforderlich sind.

Für besonders große Datenmengen, bei denen *compute()* nicht praktikabel ist, wird die Dask-Methode *persist()* eingesetzt. Im Gegensatz zu *compute()* behält *persist()* die berechneten Daten verteilt im Speicher der Worker und gibt sie nicht an den Scheduler zurück. Dies verhindert Speicherüberläufe, entlastet den Scheduler und erlaubt eine Wiederverwendung der berechneten Zwischenergebnisse in späteren Analyseschritten, etwa bei der NDVI-Berechnung in Kapitel 6.5.2. Anders als bei der Compute-Funktion wird kein konkretes NumPy-Array oder Xarray.Dataset zurückgegeben, sondern es wird unmittelbar nach dem Aufruf bereits ein Dask-Objekt zurückgegeben und die Berechnung erfolgt im Hintergrund. Um die reine Berechnungszeit eines raumzeitlichen Ausschnitts zu messen, wird bei der Nutzung von persist zusätzlich die Methode *wait()* eingesetzt, die darauf wartet, dass alle Tasks im Hintergrund abgeschlossen sind. Die Methode *persist()* wird in den Testszenarien 5 und 6 eingesetzt, wo *compute()* nicht ausreichte. Der angepasste Code lautet im Umgang mit Zarr-Dateien wie folgt:

```
    data_subset = ds['data'].sel(
                x=slice(minx, maxx),
                y=slice(maxy, miny),
                time=slice(time_point1, time_point2)
            ).persist()
    wait(data_subset)
```

Listing 13: Raumzeitliche Abfragen für Zarr-Dateien mit Dask

Für COG-Dateien erfolgt der *.persist()*-Aufruf entsprechend nach der Methode *combine_by_coords*. Da *persist()* die Übertragung der Daten vermeidet, ist die Berechnungsdauer nicht direkt mit der vom *compute()*-Befehl vergleichbar und fällt meist kürzer aus. Durch die Nutzung von *compute()* kann im Notebook direkt mit den Werten weitergearbeitet werden. Beim Aufruf von *persist()* können die Daten zwar relativ schnell aus dem Arbeitsspeicher der Worker abgefragt werden, sie liegen allerdings verteilt im System vor und nur für die Lebensdauer des Dask-Clusters. Diese Vorgehensweise zeigt die technischen Herausforderungen und Limitierungen im Umgang mit Big Data in Cloud-Umgebungen auf und liefert zugleich wertvolle Erfahrungen, wie Workarounds gestaltet sein müssen, um Formate wie Zarr und COG effizient und skalierbar zu nutzen.

6.5.2 Raumzeitliche Analysen

Dieses Kapitel behandelt die praktische Umsetzung des in Kapitel 4.3.2 beschriebenen zweiten Untersuchungsteils. Ziel ist die Durchführung raumzeitlicher NDVI-Analysen mit COG und Zarr, wobei der Fokus auf der technischen Implementierung und den Herausforderungen liegt. Dabei werden die Datensätze geöffnet, der NDVI pixelweise berechnet, raumzeitliche Mittelwerte zur jährlichen Vegetationsentwicklung ermittelt und der Anteil vegetationsreicher Flächen bestimmt. Die NDVI-Werte werden zudem gespeichert, um weitere Analysen wie pixelbasierte Modifizierungen vorzubereiten. Die geplante Trendanalyse konnte aus Zeitgründen nicht umgesetzt werden.

Berechnung des NDVI
Das Öffnen der COG- und Zarr-Dateien erfolgt analog zu Kapitel 6.5.1, wobei hier die gesamte Datei verarbeitet wird, sodass die *.sel*-Methode entfällt. Zunächst wird pixelweise der NDVI berechnet. Dafür werden die Bandinformationen für den roten und den infraroten Kanal extrahiert und in den Datentyp float32 konvertiert. Diese Umwandlung ist notwendig, da die Berechnung aufgrund von Überschreitungen des Wertebereichs von uint8 bei der Addition und Subtraktion zu fehlerhaften Ergebnissen geführt hätte. Ein kleiner Epsilon-Wert verhindert eine Division durch Null bei schwarzen Pixeln (siehe Listing 14, Zeile 6). Die Berechnung erfolgt gemäß der in Kapitel 4.3.2 beschriebenen Formel.

Für Zarr-Daten wird der NDVI als neue Variable zum Array hinzugefügt. Bei COG-Dateien wird der NDVI dagegen für jede Datei separat berechnet. Die individuellen NDVI-Arrays werden aus Performance-Gründen nicht zusammengeführt, sondern direkt weiterverarbeitet. Um sicherzustellen, dass die Zeitinformationen der ursprünglichen Dateien erhalten bleiben, werden die Metadaten des Ausgangsdatensatzes an die resultierenden NDVI-Arrays mit dem Befehl *ndvi.attrs = ds.attrs.copy()* angefügt. Die Berechnung wird durch *dask.delayed* parallelisiert, um alle Dateien effizient gleichzeitig zu verarbeiten (siehe Anhang Abschnitt D). Um den Scheduler nicht zu überlasten, wird für beide Ansätze die *persist*-Methode eingesetzt, die den NDVI für weitere Berechnungen wie Mittelwerte, Flächenanalysen oder Trendberechnungen verfügbar hält, ohne ihn erneut berechnen zu müssen.

```
1     def calculate_ndvi(ds):
2         red = ds.data.sel(band=3)
3         nir = ds.data.sel(band=4)
4         red = red.astype('float32')
5         nir = nir.astype('float32')
6         epsilon = 1e-10
7         ndvi = (nir - red) / (nir + red + epsilon)
8         ds['ndvi'] = ndvi
9         return ds
10
11    with ms_ndvi.sample(f"Experiment {experiment_id}"):
12        with performance_report(filename=f"ndviZarr/{experiment_id}.html"):
13            ds_ndvi = calculate_ndvi(ds_zarr)
14            ds_ndvi_persisted = ds_ndvi.persist()
15            wait(ds_ndvi_persisted)
```

Listing 14: Berechnung des NDVI für Zarr-Dateien

Berechnung raumzeitlicher Statistiken
Die Berechnung der Mittelwerte erfolgt mit der Methode *calculateStatistics*, wobei der Ansatz je nach Datenformat variiert. Für COG-Dateien werden die Dateien einzeln geladen, der NDVI berechnet und daraus der mittlere NDVI sowie der prozentuale Anteil vegetationsreicher Fläche pro Datei bestimmt. Der mittlere NDVI wird mit *.mean()* berechnet und die Zeitinformation aus dem Attribut *TIFFTAG_ DATETIME* wird zur Zuordnung der Ergebnisse genutzt. Die Ergebnisse (Jahr, NDVI, Vegetationsfläche) werden in einem DataFrame gesammelt und anschließend nach Jahr aggregiert, ohne die Dateien zu mergen, was den Prozess effizient hält. Die Berechnung erfolgt parallel mit *dask.delayed* und *dask.compute* erzeugt die Werte für alle Dateien (siehe Anhang Abschnitt D). Die prozentuale Fläche wird berechnet, indem ein Schwellwert von 0,25 festgelegt wird, basierend auf praktischen Erfahrungen im Untersuchungsgebiet Harz, wo NDVI-Werte zwischen 0,2 und 0,3 typischerweise als Vegetation interpretiert werden. Eine Maske markiert Pixel über dem Schwellwert und der Flächenanteil wird relativ zur Gesamtfläche berechnet.

Bei Zarr-Dateien kann der mittlere NDVI direkt für jeden Zeitpunkt mit *.mean()* berechnet werden, da die Daten bereits zeitlich organisiert sind und keine Zusammenführung erforderlich ist (siehe Anhang Abschnitt E). Anders als bei COG-Dateien erfolgt die Berechnung für alle Zeitpunkte daher automatisch. Die Berechnung der Vegetationsmaske und des Flächenanteils erfolgt analog zu COG. Da Xarray, Zarr und Dask direkt miteinander interagieren können, müssen keine zusätzlichen Dask-APIs verwendet

werden. Die Berechnungen werden durch einen einfachen Funktionsaufruf durchgeführt, wodurch der Workflow für Anwendende vergleichsweise unkompliziert bleibt.

Trendanalysen
Obwohl in dieser Implementierung nicht umgesetzt, wären Analysen wie Trendanalysen oder Histogrammbildungen wertvolle Ergänzungen gewesen, um die langfristige Vegetationsentwicklung und die Verteilung der NDVI-Werte zu untersuchen. Solche Analysen hätten vermutlich deutliche Performance-Unterschiede zwischen COG und Zarr aufgezeigt. Für COG-Dateien hätte dies jedoch eine räumliche Zusammenführung der Daten erfordert, da Trendanalysen und Histogramme typischerweise auf einem konsistenten Datensatz basieren und nicht auf fragmentierten Teilbereichen. Dies hätte erheblichen Speicher- und Rechenaufwand bedeutet und den Workflow deutlich komplexer und ressourcenintensiver gemacht. Zarr hingegen ist durch seine zusammenhängende Datenstruktur optimal für solche Analysen geeignet. Diese erweiterten Analysen hätten somit nicht nur tiefere Einblicke in die Daten ermöglicht, sondern auch die Stärken und Schwächen der beiden Formate bei komplexeren raumzeitlichen Berechnungen noch deutlicher hervorgehoben.

Speicherung
Zur persistenten Speicherung des NDVI wird die Modifizierbarkeit der Formate untersucht. Wie in Kapitel 5 beschrieben, erlauben COG-Dateien keine direkte Modifikation. Um NDVI-Werte hinzuzufügen, müsste die gesamte Datei neu geschrieben werden, z. B. durch Verknüpfung der bestehenden Bänder mit dem berechneten NDVI in einem Array:
new_cog = np.stack([band1, band2, band3, band4, ndvi_band], axis=0)
Dieses Array kann anschließend in eine neue COG-Datei geschrieben werden, wie in Kapitel 6.4.1 beschrieben. Bibliotheken wie Rioxarray oder Rio-Cogeo bieten zudem keinen Verarbeitungsmodus für pixelbasierte Änderungen. Zwar könnte Rasterio verwendet werden, jedoch riskiert dies, die COG-Struktur zu beschädigen, da Overviews und Indizes nicht automatisch aktualisiert werden. Daher wird empfohlen, Modifikationen nur durch vollständiges Neuschreiben der Datei vorzunehmen.

Bei Zarr-Dateien sind, wie in Kapitel 5 beschrieben, bestimmte Modifikationen möglich. Dafür wird die Datei mit S3fs geöffnet, der NDVI als neue Variable hinzugefügt und mithilfe von *.to_zarr()* im *float32*-Format und festgelegten Chunks gespeichert (siehe Listing 15). Der Modus *'a'* erlaubt das Hinzufügen neuer Variablen, überschreibt dabei jedoch die Dimensionskoordinaten. Die Funktion *clean_encoding* ist erforderlich, um inkompatible, Xarray-spezifische Metadaten (*_FillValue*, *add_offset*, *scale_factor*) zu entfernen, um Fehler bei der Speicherung zu vermeiden. Wenn die Variable bereits existiert oder gezielte Änderungen einzelner Werte oder Pixel vorgenommen werden sollen, muss der Modus „r+“ verwendet werden. Dieser Modus aktualisiert nur den betroffenen Chunk anstelle der gesamten Datei, was bei kleinen Änderungen erheblich effizienter ist. Der Workaround bleibt dabei der Gleiche, nur dass anstelle der *.assign*-Funktion ein einzelner Wert beispielsweise durch den folgenden Befehl verändert wird:
ds['ndvi'].loc['time': '2016-01-01', 'x': 598000.3, 'y': 5753999.7] = 0.35.

```
1
2     ds_existing = xr.open_zarr(store, mask_and_scale=False)
3     # Aktualisieren oder Hinzufuegen der Variable
4     ds_existing = ds_existing.assign({variable_name: ds})
5     for var in ds_existing.variables:
6         ds_existing[var] = clean_encoding(ds_existing[var])
7     mode = 'a'    # 'r+' fuer Pixelaenderungen
8     ds_existing.to_zarr(store=store, mode=mode, encoding=encoding_interp,
    consolidated=True)
```

Listing 15: Speicherung der NDVI-Werte in der Zarr-Datei

7 Analyse cloud-optimierter Datenformate für raumzeitliche Rasterdaten

In diesem Kapitel werden die Ergebnisse der praktischen Evaluierung der cloud-optimierten Datenformate hinsichtlich ihrer Eignung für die Speicherung und Analyse raumzeitlicher Rasterdaten präsentiert. Dafür werden die Testmessungen gemäß der in Kapitel 6.4 und 6.5 beschriebenen Vorgehensweise durchgeführt. Die Ergebnisse werden anschließend hinsichtlich der festgelegten Metriken wie Laufzeiten, Dateigrößen und Datenmengen ausgewertet und anhand von grafischen Darstellungen und statistischen Analysen veranschaulicht.

7.1 Speicherung in der Cloud

In diesem Abschnitt werden die Ergebnisse der Untersuchung zur Erstellung und zum Upload von Zarr- und COG-Dateien vorgestellt. Ziel der Analyse ist es, die Performance, Stabilität und Skalierbarkeit beider Formate bei variierenden Datengrößen, Partitionen und Kompressionsmethoden zu bewerten. Die gewonnenen Erkenntnisse dienen dazu, fundierte Empfehlungen für den praktischen Einsatz beider Formate abzuleiten.

Performance:
Bei kleineren Datensätzen, wie dem Upload von einem DOP, zeigt Zarr tendenziell schnellere Uploadzeiten im Vergleich zu COG, insbesondere bei optimierten Kachelgrößen. Beispielsweise benötigte Zarr beim Upload eines einzelnen DOPs mit einer Kachelgröße von 1.024 x 1.024 Pixeln durchschnittlich 9,44 Sekunden, während COG unter denselben Bedingungen 26,76 Sekunden benötigte (siehe Tabelle 8). Bei größeren Datensätzen, wie dem Upload von 54 oder 100 DOPs, wird der Performance-Vorteil von Zarr weniger deutlich. In einigen Fällen übertrifft COG sogar die Upload-Geschwindigkeit von Zarr. Zum Beispiel benötigte COG beim Upload von 54 DOPs mit einer Kachelgröße von 1.024 x 1.024 Pixeln durchschnittlich 175,86 Sekunden, während Zarr 233,09 Sekunden benötigte.

	1 DOP	**9 DOPs**	**54 DOPs**	**100 DOPs**
Zarr	9,44	37,80	233,09	452,64
COG	26,76	45,62	175,86	312,69

Tabelle 8: Uploadzeiten in Sekunden bei einer Kachelgröße von 1.024 x 1.024 Pixeln

Skalierbarkeit:
Zarr ist mit optimaler Kachelgröße und Komprimierung effizienter, insbesondere bei großen Datenmengen. Die Upload-Zeiten steigen bei beiden Formaten mit der Datenmenge, jedoch zeigt Zarr eine bessere Skalierbarkeit. Bei Zarr steigt die Upload-Zeit nicht immer proportional zur Datenmenge, was auf eine effizientere Datenverarbeitung hindeutet. Beispielsweise zeigt Zarr bei einer Kachelgröße von 10.000 x 10.000 Pixeln einen sublinearen Anstieg der Upload-Zeiten, während COG in ähnlichen Szenarien eher einen linearen Anstieg aufweist (siehe Tabelle 9).

	1 DOP	**9 DOPs**	**54 DOPs**	**100 DOPs**
Zarr	18,97	35,47	129,01	227,90
COG	37,94	59,40	589,95	1027,00

Tabelle 9: Uploadzeiten in Sekunden bei einer Kachelgröße von 10.000 x 10.000 Pixeln

Robustheit:
Bei größeren Datensätzen zeigt Zarr tendenziell größere Schwankungen in den Upload-Zeiten im Vergleich zu COG. Zum Beispiel betrug die Standardabweichung bei Zarr für den Upload von 54 Bildern

mit einer Kachelgröße von 4.096 Pixeln 24,83 Sekunden, während sie bei COG unter denselben Bedingungen nur 7,25 Sekunden betrug. Dies deutet darauf hin, dass COG bei größeren Datenmengen eine höhere Stabilität und Robustheit aufweist. Die Wahl der Kachelgröße und Kompression zeigt insgesamt einen erheblichen Einfluss auf die Performance. Die folgenden Analysen untersuchen den Einfluss unterschiedlicher Kompressionsalgorithmen und Kachelgrößen auf die Performance beider Formate daher im Detail.

7.1.1 Einfluss der Datenpartitionierung

Die effiziente Verarbeitung von Daten in parallelen Systemen wie Dask erfordert eine sorgfältige Auswahl der Partitionierungsgrößen, um sowohl die Speichernutzung als auch die Rechenleistung zu optimieren. Große Partitionen und übermäßig komplexe Abhängigkeitsgraphen können dabei zu erheblichen Leistungseinbußen führen. Um die verfügbare Speicherkapazität eines jeden Rechenknotens optimal auszunutzen, sollten die Datenpartitionen so gewählt werden, dass mehrere Partitionen gleichzeitig in den Arbeitsspeicher eines Knoten passen. Gleichzeitig sollte jedoch auch vermieden werden, dass die Chunk-Größe zu klein gewählt wird, da dies zu einer ineffizienten Nutzung der Rechenressourcen führen kann. Eine zu kleine Chunk-Größe erhöht den Verwaltungsaufwand des Systems erheblich, da der Scheduler eine große Anzahl von Aufgaben koordinieren muss (Anaconda, Inc. and Contributors, 2018c, o.S.).

Für die DOPs des LGLN zeigt diese Untersuchung, dass die Datenpartition nicht nur die Verarbeitungsgeschwindigkeit erheblich beeinflusst, sondern auch die resultierende Dateigröße. Bei Einzeldateien (1 DOP) bleibt die Dateigröße von COG- und Zarr-Dateien unabhängig von der Kachelgröße nahezu identisch. Beispielsweise beträgt die Dateigröße bei einer Kachelgröße von 256 × 256 Pixeln für beide Formate etwa 419 MB. Die Ergebnisse in Tabelle 10 zeigen jedoch, dass die Dateigröße mit zunehmender Kachelgröße tendenziell ansteigt. So erreicht sie bei einer Kachelgröße von 8.192 × 8.192 Pixeln etwa 1073 MB. Beim gleichzeitigen Upload mehrerer COG-Dateien skaliert die Dateigröße linear, wobei die Größe pro Bild unverändert bleibt. Das bedeutet, dass bei 100 DOPs mit der Standard-Kachelgröße von 512 × 512 Pixeln etwa 42 GB Speicherplatz benötigt werden. Zarr-Dateien skalieren bei wachsender Datenmenge effizienter. Insbesondere bei größeren Kachelgrößen wird der Unterschied der Dateistrukturen deutlich. Während COG-Dateien bei 8.192 × 8.192 Pixeln einen Speicherbedarf von 107 GB aufweisen, benötigen Zarr-Dateien mit 52 GB fast nur die Hälfte. Zudem zeigt sich, dass durch die Wahl einer geeigneten Kachelgröße die resultierende Dateigröße erheblich reduziert werden kann. Insbesondere Kachelgrößen, die durch 16 teilbar sind und ein Bruchteil der Ausgangsgröße von 10.000 × 10.000 Pixeln darstellen, erzeugen geringere Dateigrößen. Dazu zählen Kachelgrößen wie 400 × 400, 2.000 × 2.000 und 10.000 × 10.000 Pixel. In diesen Fällen bleibt die Dateigröße sowohl bei COG als auch bei Zarr bei etwa 400 MB. Dies deutet darauf hin, dass solche Kachelgrößen eine besonders effiziente Balance zwischen Speicherbedarf und Performance bieten. Für COGs und Zarr empfiehlt es sich daher, die Kachelgröße nicht nur an den Anwendungszweck, sondern auch an die Größe der Ausgangsdaten anzupassen.

Kachelgröße	**Dateigröße 1 COG [MB]**	**Dateigröße Zarr (1DOP) [MB]**	**Dateigröße 100 COGs [GB]**	**Dateigröße Zarr (100 DOPs) [GB]**
256x256	419,46	419,60	-	-
400x400	400,01	400,17	40,00	-
512x512	419,44	419,60	41,94	-
1.024x1.024	419,43	419,60	41,94	40,28
2.000x2.000	400,00	400,17	40,00	40,00
4.096x4.096	603,98	604,15	60,40	45,37
8.192x8.192	1073,74	1073,91	107,37	52,61
10.000x10.000	400,00	400,17	40,00	40,00

Tabelle 10: Speicherbedarf von COG- und Zarr-Dateien in Abhängigkeit der Kachelgröße und Datenmenge

Die Uploadzeit für COGs zeigt einen parabelförmigen Verlauf (siehe Abbildung 35). Bei kleinen Kachelgrößen sind die Uploadzeiten hoch, nehmen jedoch mit zunehmender Kachelgröße zunächst deutlich ab. Die geringste Uploadzeit wird bei einer Kachelgröße von 1.024 × 1.024 erreicht, bevor die Zeiten mit größeren Kachelgrößen wieder ansteigen. Dieser Verlauf ist besonders bei mittleren und großen Datenmengen gut erkennbar. Bei kleineren Datenmengen (1 oder 9 DOPs) bleibt die Uploadzeit hingegen unabhängig von der Kachelgröße relativ konstant. Im Gegensatz dazu zeigt das Zarr-Format keinen parabelförmigen Verlauf (siehe Abbildung 36). Die Uploadzeit verringert sich kontinuierlich mit zunehmender Chunkgröße und stagniert schließlich auf einem konstanten Niveau. Kleine Chunkgrößen führen bei Zarr jedoch nicht nur zu langen Laufzeiten, sondern erschweren auch die Durchführung von Uploads, besonders bei größeren Datenmengen. Bei einer Kachelgröße von 1.024 × 1.024 Pixeln sind die Uploadzeiten für Zarr mit etwa 460 Sekunden deutlich länger als die Uploadzeiten einzelner COG-Dateien mit 312 Sekunden. Darüber hinaus war die Durchführung bei mittleren und großen Datenmengen mit Zarr mit den Kachelgrößen bis 512 x 512 nicht möglich. Ähnliche Schwierigkeiten traten bei COG auf, insbesondere bei der Kachelgröße 256 × 256 für große Datenmengen. Fehlende Werte in den Ergebnissen für 1 DOP bei COG und 54 DOP bei Zarr sind auf unkonstante Laufzeiten und Netzwerkschwankungen zurückzuführen und werden deshalb aus der Analyse ausgeschlossen.

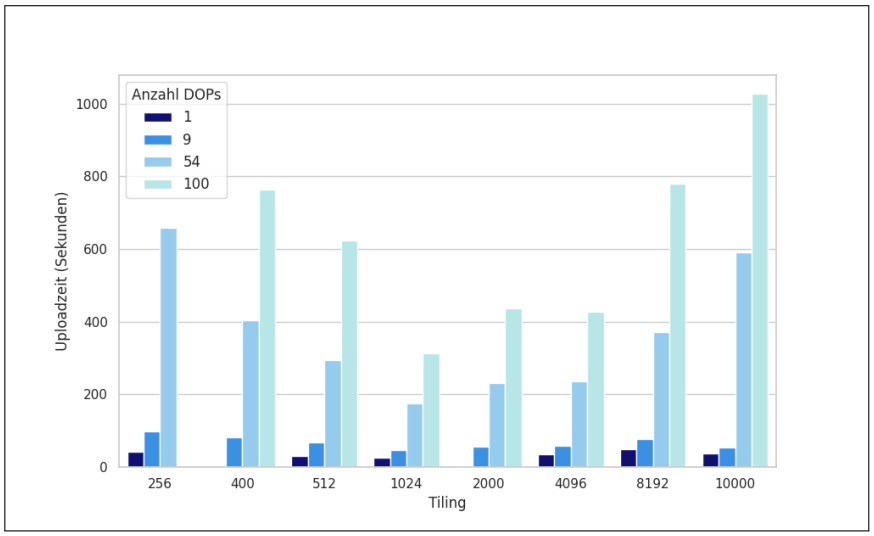

Abbildung 35: Uploadzeiten von COG-Dateien in Abhängigkeit des Tilings

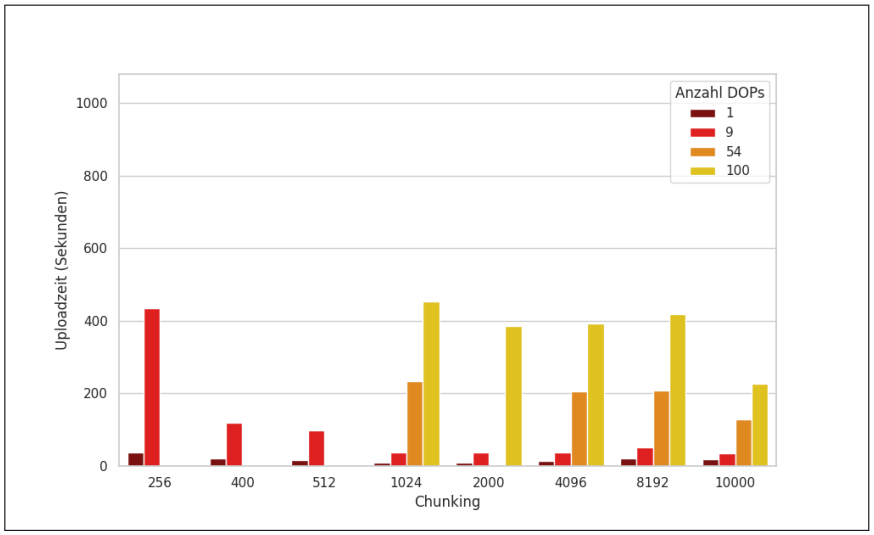

Abbildung 36: Uploadzeiten von Zarr-Dateien in Abhängigkeit des Chunkings

Bei großen Kachelgrößen zeigt Zarr seine Stärke. Die Uploadzeiten stabilisieren sich und betragen für 100 DOPs bei 10.000 x 10.000 Pixeln nur etwa 230 Sekunden, was fast 80 % weniger als bei COG ist. Zarr profitiert hier von seiner effizienten Handhabung großer, zusammenhängender Datenblöcke. Darüber hinaus skaliert Zarr effizient mit der Anzahl der DOPs, da die Uploadzeit mit zunehmender Kachelgröße weiter sinkt und bei 10.000 × 10.000 Pixeln besonders effizient wird. Im Gegensatz dazu ist die Skalierung bei COG weniger effizient. Mit steigender Anzahl der DOPs und größeren Kachelgrößen steigen die Uploadzeiten bei COG an, was es bei großen Datenmengen weniger konkurrenzfähig macht. Bei kleinen Datenmengen ist die Uploadzeit bei beiden Formaten relativ konstant.

Die Unterschiede zwischen den Formaten werden auch durch die CPU-Auslastung verdeutlicht. Bei Zarr zeigt die CPU-Auslastung bei kleineren Chunkgrößen eine deutlich größere Streuung (siehe Anhang Abschnitt C). Zwischen den Chunkgrößen 1.024 × 1.024 und 2.000 × 2.000 Pixeln tritt eine markanter Senkung der CPU-Auslastung auf, was die Problematik kleiner Chunkgrößen bei größeren Datenmengen unterstreicht (siehe Abbildung 44). Ähnlich zeigt auch COG eine höhere CPU-Auslastung bei kleinen Kachelgrößen, die jedoch mit zunehmender Kachelgröße kontinuierlich abnimmt (siehe Abbildung 43). Bei Betrachtung des Durchsatzes, definiert als das Verhältnis von Dateigröße zu Uploadgeschwindigkeit, zeigt sich, dass dieser mit zunehmender Kachelgröße beim Upload von COG-Dateien steigt. Ein höherer Durchsatz bedeutet, dass mehr Daten in kürzerer Zeit verarbeitet oder hochgeladen werden können. Kachelgrößen die besser zur Datenstruktur der DOPs passen und einen geringeren Speicherbedarf haben, weisen jedoch einen geringeren Durchsatz auf. Insbesondere bei einer Kachelgröße von 10.000 x 10.0000 wird ein sehr geringer Durchsatz erreicht, da der Upload im Verhältnis zur Dateigröße sehr lange dauert. Für den praktischen Einsatz empfiehlt es sich daher, Kachelgrößen zu wählen, die eine gute Balance zwischen Dateigröße, Uploadzeit, CPU-Auslastung und Speicherbedarf bieten. Kachelgrößen von 1.024 x 1.024 oder 2.000 x 2.000 bieten eine optimale Konfiguration, die sowohl Effizienz als auch Performance gewährleistet. Zarr hingegen zeigt mit zunehmender Kachelgröße einen kontinuierlich steigenden Durchsatz. Der höchste Durchsatz wird bei großen Datenmengen und großen Kachelgrößen erzielt, was Zarrs Eignung für Big-Data-Szenarien unterstreicht. Aufgrund seiner flexibleren Spezifikation der Partitionierung wird für Zarr größere Chunkgrößen wie 10.000 × 10.000 Pixel empfohlen, um einen hohen Durchsatz und eine bessere Performance zu gewährleisten.

7.1.2 Einfluss der Dateikompression

Die effiziente Speicherung und der Upload großer Datenmengen in die Cloud sind entscheidende Faktoren für die Leistungsfähigkeit moderner Datenverarbeitungssysteme. In diesem Kontext spielt die Dateikompression eine zentrale Rolle, um sowohl den Speicherbedarf zu minimieren als auch die Übertragungszeiten zu optimieren. Diese Untersuchung fokussiert sich auf die Kompression und Upload-Performance von Zarr- und COG-Dateien unter Verwendung verschiedener verlustfreier Kompressionsalgorithmen wie LZ4, ZSTD, Deflate, LZW und Blosc. In diesem Abschnitt werden die Ergebnisse hinsichtlich der Kompressionsrate, Dateigrößenreduktion und des Datendurchsatzes (MB/s) bewertet. Darüber hinaus wird der Einfluss der Kachelgröße im Zusammenhang mit verschiedenen Kompressionsmethoden evaluiert.

Für COG-Dateien werden die Kompressionsmethoden LZW, Deflate und ZSTD betrachtet. Die resultierenden Dateigrößen und Uploadzeiten werden jeweils mit der unkomprmierten Dateigröße verglichen. Diese ist bei COG-Dateien mit der Einstellung „Raw “ versehen. Die Deflate-, LZW- und ZSTD-Algorithmen unterstützen zudem die Verwendung von Prädiktoren. Die Ergebnisse zeigen, dass die Kompression mit dem Prädiktor 2 insgesamt geringere Dateigrößen hervorbringt (siehe Abbildung 37). Die LZW-Komprimierung mit dem Prädiktor 1 ist ineffizient, da sie die durchschnittlichen Dateigrößen gegenüber der unkomprimierten Datei (Raw_1) vergrößert. Die geringsten Dateigrößen können mit dem Deflate- und dem ZSTD-Algorithmus erzielt werden. Auffällig ist außerdem, das mit zunehmender Datenmenge die gleichzeitig geschrieben wird, die durchschnittliche Dateigröße sinkt. Dies deutet darauf hin, dass einige DOPs besser komprimiert werden können als andere. Insbesondere bei der Speicherung der Differenzen mit dem Prädiktor 2 sinkt dann die Dateigröße bei einzelnen DOPs mehr als bei

anderen, da sie ähnlichere Werte enthalten. Mit Kompression ist die Dateigröße mit kleiner Kachelung in der Regel geringer als mit größeren Kacheln. Insgesamt sind die Unterschiede in den resultierenden Dateigrößen bei unterschiedlichen Kachelgrößen eher gering und zu vernachlässigen.

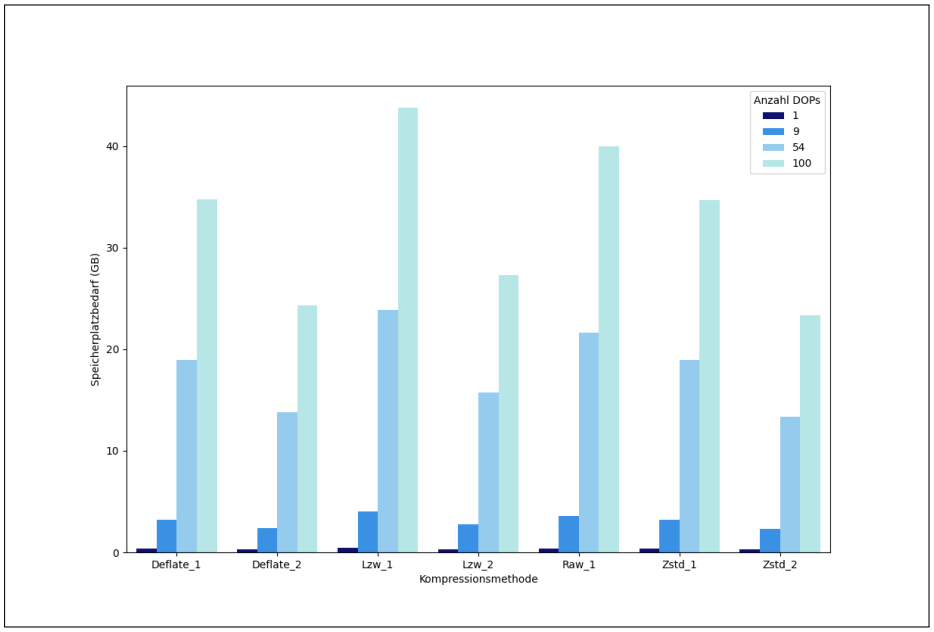

Abbildung 37: Speicherplatzbedarf von COG-Dateien in Abhängigkeit der Kompressionsmethode

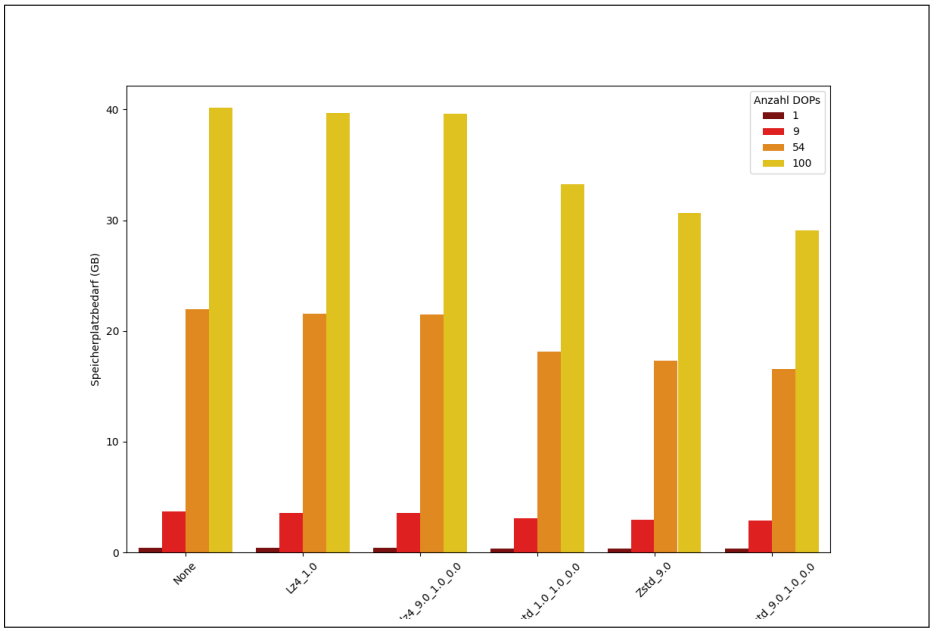

Abbildung 38: Speicherplatzbedarf von Zarr-Dateien in Abhängigkeit der Kompressionsmethode (Auswahl)

Die Analyse der Zarr-Dateigrößen zeigt, dass der ZSTD-Algorithmus im Vergleich zum LZ4-Algorithmus insgesamt eine größere Reduktion der Dateigrößen ermöglicht, insbesondere in Kombination mit dem Blosc-Kompressor (siehe Abbildung 38). Bei größeren Datenmengen von bis zu 40 GB (100 DOPs) kann ZSTD eine Dateigrößenreduktion von bis zu 25% erreichen, während LZ4 weniger als 10% Einsparung erzielt. Der Einfluss des Kompressionsgrades ist dabei für den ZSTD-Algorithmus ausschlaggebender als der Parameter Shuffling. Die kleinsten Dateigrößen werden durch eine Kombination aus ZSTD, Blosc, einem Kompressionslevel von 9 und aktivem Shuffle erreicht. Trotz der höheren Effizienz von Blosc

bei der Kompression bleibt das COG-Format führend in der Reduktion der Dateigröße. So beträgt die kleinste Dateigröße in dieser Untersuchung für 100 DOPs 28,7 GB bei Zarr, während COG mit 23,4 GB einen deutlichen Vorteil aufweist. Die Kompressionsraten in Tabelle 11 zeigen, dass das COG-Format eine maximale Kompressionsrate von 1,78:1 erreicht, während das Zarr-Format mit 1,41:1 etwas darunterliegt. Allerdings kann ein ungeeigneter Algorithmus zu größeren Dateigrößen führen. Dieser Effekt wird insbesondere bei COG-Dateien mit dem LZW-Algorithmus festgestellt.

Anzahl DOPs	**Max. KR COG**	**Max. KR Zarr**	**Min. KR COG**	**Min. KR Zarr**
1	1.48:1 (*Deflate (Pred. 2)*	1.26:1 (*Blosc (ZSTD, Lev.:9, Shuffle)*)	0.90:1 (*LZW*)	0.99:1 (*Lz4(acc.:10.0)*)
9	1.59:1 (*ZSTD (Pred. 2)*)	1.34:1 (*Blosc (ZSTD, Lev.:9, Shuffle)*)	0.89:1 (*LZW*)	0.99:1 (*Lz4(acc.:10.0)*)
54	1.67:1 (*ZSTD (Pred. 2)*)	1.36:1 (*Blosc (ZSTD, Lev.:9, Shuffle)*)	0.91:1 (*LZW*)	0.99:1 (*Lz4(acc.:10.0)*)
100	1.78:1 (*ZSTD (Pred. 2)*)	1.41:1 (*Blosc (ZSTD, Lev.:9, Shuffle)*)	0.91:1 (*LZW*))	1:1 (*Lz4(acc.:10.0)*)

Tabelle 11: Maximale und minimale Kompressionsraten (KR) nach Datenmenge und Dateiformat

Die Uploadzeiten zeigen, dass Kompression bei COG-Dateien grundsätzlich zu einer minimalen Verlängerung führt (siehe Abbildung 39). Bei kleinen und mittleren Datenmengen ist die Erhöhung der Uploadzeit marginal, bei großen Datenmengen hingegen kann sie um bis zu 20% zunehmen. Besonders betroffen sind die Algorithmen Deflate und ZSTD mit Prädiktor 2, die zwar die höchsten Kompressionsraten erzielen, jedoch auf Kosten der Uploadgeschwindigkeit. Beim Zarr-Format sind die Uploadzeiten vor allem bei der Blosc-Kompression mit ZSTD und einem Kompressionsgrad von 9 erhöht (siehe Abbildung 40). Hier verlängert sich der Upload um bis zu 60%, verglichen mit unkomprimierten Daten. Dennoch bleibt der Upload von Zarr-Dateien auch bei diesen Einstellungen effizienter als der von COG-Dateien. Zu beachten ist, dass die Ergebnisse auf einer Kachelgröße von 10.000 × 10.000 basieren, die die Effizienz von Zarr begünstigt.

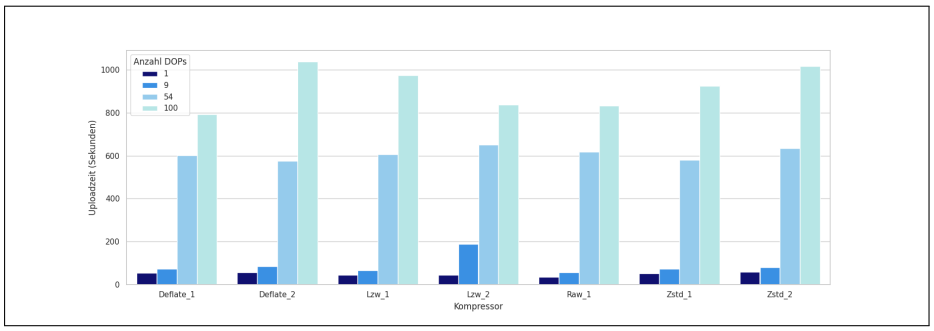

Abbildung 39: Uploadzeiten von COG-Dateien in Abhängigkeit der Kompressionsmethode

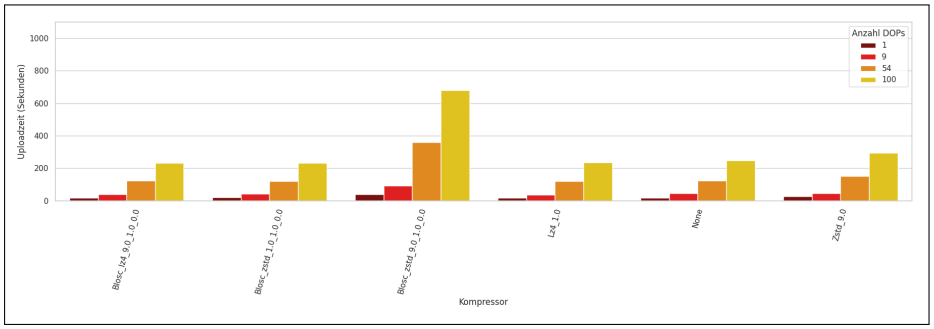

Abbildung 40: Uploadzeiten von Zarr-Dateien in Abhängigkeit der Kompressionsmethode

Ein ausgewogener Kompromiss zwischen Dateigröße und Uploadgeschwindigkeit könnte bei Zarr-Dateien die Verwendung des Blosc-Kompressors in Kombination mit dem ZSTD-Algorithmus und einem niedrigen Kompressionsgrad (z. B. Level 1) darstellen. Alternativ könnte auch der reine ZSTD-Algorithmus

ohne Blosc eingesetzt werden, um die Uploadzeiten weiter zu optimieren, ohne dabei die Kompressionsrate signifikant zu reduzieren. Für COG-Dateien bieten zwar der Deflate- und ZSTD-Algorithmus die besten Kompressionsraten, doch gehen diese mit deutlich längeren Uploadzeiten einher, insbesondere bei großen Datenmengen. Ein praktikabler Kompromiss wäre der LZW-Algorithmus mit Prädiktor 2, der zwar etwas geringere Dateireduktionen erzielt, jedoch durch seine verhältnismäßig schnellen Uploadzeiten überzeugt und eine effiziente Verarbeitung ermöglicht. Insgesamt sollte bei der Wahl der Kompressionsmethode jedoch in Abhängigkeit des Anwendungsfalls eine Abwägung zwischen Uploadgeschwindigkeit und Kompressionsrate erfolgen.

7.2 Analyse raumzeitlicher Rasterdaten

In diesem Abschnitt wird der zweite Teil des Untersuchungskonzeptes zur Analyse raumzeitlicher Rasterdaten betrachtet. Trotz begrenzter Datenpunkte konnten erste Erkenntnisse zur Effizienz und Skalierbarkeit der Formate gewonnen werden, die wichtige Hinweise für den Einsatz in praxisnahen Anwendungen liefern.

7.2.1 Performance raumzeitlicher Abfragen

In dieser Untersuchung konnten nur die Laufzeiten beim Zugriff auf Zarr-Dateien analysiert werden. Die Anzahl der HTTP-Requests, die Datenmenge und die Anzahl der Tasks wurden deshalb nicht weiter betrachtet, da keine Vergleichswerte von COG-Dateien vorliegen. Zu den zeitlichen Zugriffen konnten nur wenige Erkenntnisse gewonnen werden, da die Daten nicht unterschiedlich partitioniert werden konnten und nur wenige Zeitpunkte zur Verfügung standen.

	Räumlich	**Zeitlich**	**Räumlich mit Schnitt**	**Raumzeitlich (Klein)**	**Raumzeitlich (Groß)**
Testnummer	**7**	**8**	**9**	**10**	**11**
Kleine Datenmenge	7,51	6,30	6,62	14,68	–
Mittlere Datenmenge	5,36	5,28	6,02	15,10	44,88
Große Datenmenge	6,00	5,76	6,58	16,59	46,34

Tabelle 12: Laufzeiten für verschiedene raumzeitliche Abfragen von Zarr-Dateien in Abhängigkeit der Datenmenge und des Abfragebereichs in Sekunden

Einfluss der Partitionierung:
Kleine Chunks ermöglichen zwar präzisere Zugriffe auf kleinere Datenbereiche, führen jedoch zu einer höheren Anzahl von HTTP-Requests und damit zu längeren Zugriffszeiten. So dauerte die Abfrage eines kleinen Bereichs zu einem Zeitpunkt bei einer Chunkgröße von 512×512 Pixeln etwa 7 Sekunden. Bei einer größeren Chunkgröße von 10.000×10.000 Pixeln verkürzte sich die Zugriffszeit auf nur 4,16 Sekunden (siehe Tabelle Anhang Abschnitt F). Große Chunks reduzieren die Anzahl der notwendigen HTTP-Requests, können jedoch bei kleineren Abfragen die Ladezeiten erhöhen, da mehr Daten als benötigt übertragen werden. In den durchgeführten Tests zeigte sich ein parabelförmiger Verlauf der Zugriffszeit in Abhängigkeit von der Chunkgröße. Mit zunehmender Chunkgröße sank die Ladezeit zunächst und erreichte ein Minimum bei einer Chunkgröße von etwa 2.000 Pixeln. Danach stieg die Zugriffsdauer wieder an. Eine mittlere Chunkgröße von etwa 2.000 Pixeln bietet somit ein optimales Gleichgewicht zwischen I/O-Overhead und Datenmenge pro Chunk. Mit COG-Dateien konnte dies zwar nicht umfassend untersucht werden, jedoch zeigen Stichproben ähnliche Ergebnisse.

Einfluss der Kompression:
Die verschiedenen Kompressionsmethoden führten zu Schwankungen in den Zugriffszeiten von bis zu 15 Sekunden (siehe Tabelle Anhang Abschnitt F). Interessanterweise verbesserte die Kompression oftmals die Zugriffszeit. Die Zugriffszeit ohne Kompression war fast immer länger als die mit Kompression. Beispielsweise dauerte die Abfrage des großen Bereichs bei drei Zeitpunkten ohne Kompression 44,45 Sekunden, während sie mit dem Blosc-Kompressor und dem ZSTD-Algorithmus nur 30,38 Sekunden benötigte. Der ZSTD-Kompressionsalgorithmus erzielte bessere Laufzeiten als die LZ4-Kompression.

Besonders effektiv waren der Blosc-Kompressor mit ZSTD bei Kompressionslevel 9 sowie die einfache ZSTD-Kompression mit Level 1. Dies unterstreicht die Bedeutung der Wahl des richtigen Kompressionsalgorithmus und der Kompressionsstufe für die Performance.

Einfluss der Datenmenge:
Die Größe des Datacubes hatte zunächst einen geringen Einfluss auf die Zugriffszeiten. In den Testfällen 7 bis 11 zeigte sich, dass ähnliche Bereiche unabhängig von der Dateigröße ähnlich schnell abgefragt werden konnten (siehe Tabelle 12). Obwohl dieser Aspekt mit COG-Dateien nicht getestet wurde, ist aufgrund der höheren Anzahl an HTTP-Requests zu erwarten, dass die Anzahl der Dateien dort einen Unterschied macht.

Die Größe des Abfragebereichs hat dagegen einen großen Einfluss auf die Zugriffszeiten. Mit zunehmender Größe des Abfragebereichs und der Datenmenge erhöhte sich die Dauer der Zugriffszeit. Während die Abfrage eines kleinen Testbereichs nur etwa 5 Sekunden dauerte, benötigte das Laden der Daten für den Testfall 11 mit dem größten Bereich bis zu 46 Sekunden (siehe Tabelle 12). Die Dimension — ob räumlich oder zeitlich — hatte keinen wesentlichen Einfluss auf die Zugriffszeit. Sowohl die Abfrage von zwei Datensätzen zu einem Zeitpunkt als auch die Abfrage von einem Datensatz zu zwei Zeitpunkten dauerten ähnlich lang.

7.2.2 Performance raumzeitlicher Analysen

Im Rahmen des zweiten Teils der Analyse wurde demonstriert, dass sowohl mit COG- als auch mit Zarr-Dateien raumzeitliche Analysen erfolgreich durchgeführt werden können. Die Ergebnisse der NDVI-Berechnungen und der raumzeitlichen Mittelwertbildung stimmen überein und zeigten einen signifikanten Rückgang der Vegetation im Harz zwischen 2016 und 2022. In Testausschnitten zeigte sich ein Rückgang von über 75% gesunder Vegetation.

Bei der Berechnung erwies sich eine mittlere Partitionsgröße als optimal für die Performance. Stichproben zeigen, dass die Berechnung mit einer Kachelgröße von 1.024×1.024 Pixeln mit Zarr-Dateien nur 29 Sekunden betrug, während sie mit einer größeren Kachelgröße für kleine Datenmengen 40 Sekunden benötigte. Für mittlere Datenmengen betrug die Berechnungszeit etwa 90 Sekunden. Diese Ergebnisse deuten darauf hin, dass die Wahl der Partitionsgröße einen erheblichen Einfluss auf die Effizienz der Datenverarbeitung hat. Nachdem der NDVI einmal berechnet war, konnten räumliche Statistiken sehr schnell abgeleitet werden. Die Berechnung der NDVI-Reduktion konnte in wenigen Sekunden (etwa 1–2) durchgeführt werden. Dies unterstreicht die Eignung der Formate für schnelle und umfassende Analysen.

Aufgrund der zeitintensiven Implementierung war es nicht möglich, die Analyse vollständig abzuschließen und konkrete, vergleichbare Messwerte zwischen den Formaten zu erzielen. Dennoch liefern die Implementierung und erste Stichproben wertvolle Einsichten in die Leistungsfähigkeit der beiden Formate. Zarr bietet durch seine flexible Datenstruktur und die enge Integration mit modernen Analysewerkzeugen wie Dask und Xarray klare Vorteile in Performance und Handhabung bei großen Datensätzen. Es ist zu vermuten, dass bei weitergehender Optimierung und vollständiger Durchführung der Analysen Zarr aufgrund seiner effizienten Datenstruktur und besseren Modifizierbarkeit gegenüber COG noch größere Vorteile in raumzeitlichen Analysen aufweisen wird. Bei der Berechnung raumzeitlicher Statistiken können mit COG mit wenigen Zeitpunkten vergleichbare Ergebnisse erzielt werden. Bei einer vollständigen Implementierung insbesondere bei rechenintensiven Trendanalysen und prädiktiven Modellen kann Zarr deutliche Vorteile in der Geschwindigkeit und Skalierbarkeit gegenüber COG aufweisen, sofern dies eine Zusammenführung der einzelnen Datensätze erfordern würde.

8 Evaluierung

In diesem Kapitel werden die Ergebnisse aus dem theoretischen Vergleich und der praktischen Implementierung ausgewertet und zusammengefasst. Zunächst erfolgt eine Reflexion der eingesetzten Architektur und Datengrundlage. Auf dieser Basis werden Schlussfolgerungen für den Einsatz der cloud-optimierten Formate im Geodatenmanagement sowie deren Eignung für die Speicherung und Analyse raumzeitlicher Rasterdaten abgeleitet.

8.1 Bewertung der verwendeten Architektur

Die entwickelte Architektur bietet eine skalierbare und flexible Lösung zur Analyse von COG- und Zarr-Dateien im Kontext raumzeitlicher Rasterdaten. Durch die Integration moderner Technologien wie Kubernetes, Dask und Xarray ermöglicht sie die effiziente Verarbeitung großer Datenmengen. Die Nutzung von Kubernetes als Orchestrierungsplattform ermöglicht die dynamische Skalierung der Ressourcen je nach Anforderung, wodurch Pods für den Dask-Scheduler und Dask-Worker dynamisch bereitgestellt werden können. Durch Dask wird dabei eine parallele und verteilte Verarbeitung ermöglicht, um die Vorteile der Partitionierung beider Formate evaluieren zu können. Xarray erleichtert die Handhabung von Daten in Form von beschrifteten Arrays, die über Dask effizient aufgeteilt und verarbeitet werden können. Die Anbindung an JupyterHub schafft eine interaktive und benutzerfreundliche Umgebung, die eine flexible Nutzung und einfache Erweiterbarkeit der Architektur gewährleistet. Die Architektur ist modular aufgebaut, sodass neue Technologien oder Anforderungen problemlos integriert werden können. Die Wahl von Open-Source-Technologien macht die Architektur leistungsstark, kostengünstig und bietet eine effiziente Lösung für die Speicherung und Analyse raumzeitlicher Rasterdaten.

Besonders positiv hervorzuheben ist die Zusammenarbeit von Dask, Xarray und Zarr, die eine moderne und performante Datenverarbeitung ermöglicht. Dies verdeutlicht, dass Zugänglichkeit nicht allein durch Formate, sondern vor allem durch die gezielte Nutzung von Synergien und die enge Integration verschiedener Technologien erreicht wird. Vorreiter wie das Pangeo-Projekt unterstreichen diesen ganzheitlichen Ansatz und treiben die Weiterentwicklung entscheidend voran. Die Implementierung im LGLN zeigte, dass die Architektur flexibel genug ist, um auch in anderen wissenschaftlichen Kontexten effizient eingesetzt zu werden. Die Kombination mit JupyterHub bot eine vertraute und interaktive Entwicklungsumgebung, die es ermöglichte, den „code-to-data“-Ansatz zu erproben.

Gleichzeitig werden jedoch die Herausforderungen einer derart komplexen Architektur deutlich, die viele Unbekannte auf unterschiedlichen Ebenen mit sich bringt. Probleme mit Kubernetes (z. B. Dask-Cluster auf falschen Pods, unterschiedliche Ressourcenverfügbarkeit im Mehrbenutzerbetrieb, nicht gelöschte Dask-Cluster), Dask (Scheduler-Limits, Task-Overheads, Wahl der richtigen API, schwieriges Debugging) und der Cloud-Umgebung (Netzwerkprobleme, Schwierigkeiten bei der Installation bestimmter Bibliotheken wie GDAL oder Graphviz, unzureichende Dokumentation) führten zu einem hohen Zeitaufwand. Besonders herausfordernd war es, Fehlerquellen zu identifizieren und Zusammenhänge zu verstehen, um gezielt Lösungen zu finden. Dask erwies sich als leistungsfähiges, aber äußerst komplexes Tool. Die Konfiguration des Clusters und die Nutzung der APIs erforderten ein tiefgehendes Verständnis, insbesondere da Fehler wie der Scheduler-Absturz oft nur durch unspezifische Fehlermeldungen wie „Connection closed“ sichtbar wurden. Tools wie das Dask-Dashboard und der Task-Graph erwiesen sich dabei als hilfreiche Debugging-Methoden.

Darüber hinaus erfordern Big-Data-Szenarien oft einen abweichenden Workflow im Vergleich zur Verarbeitung kleinerer Datenmengen mit traditionellen Methoden. In dieser Arbeit wird deutlich, dass bei großen Datenmengen alternative Ansätze wie der Einsatz der Methode persist() oder die verteilte Verarbeitung und Speicherung notwendig sind. Solche Workflows sind weniger transparent, schwerer nachzuvollziehen und erfordern ein tieferes technisches Verständnis, da Zwischenschritte und Datenflüsse nicht unmittelbar sichtbar sind. Die Anpassung an diese neuen Methoden und Denkweisen stellt einen Lernprozess dar, der Zeit und Erfahrung erfordert.

Die komplexe Architektur integriert viele Technologien und Bibliotheken, was fundiertes Wissen über deren Unterschiede und Handhabung voraussetzt. Beispielsweise zeigte sich, dass sich Xarray und Rioxarray im Umgang mit Datentypen erheblich unterscheiden. Zudem ist die spezifische Behandlung von Nullwerten in Xarray aus Anwendersicht ein wichtiger Aspekt, um fehlerhafte Analysen zu vermeiden. Eine umfassende Dokumentation der eingesetzten Technologien ist daher unerlässlich.

Die hochauflösenden DOPs stellten in Verbindung mit der Architektur eine besondere Herausforderung dar. Während der Workflow in der Forschung häufig mit weniger hochauflösenden Satellitendaten angewendet wird, zeigte sich, dass die Kombination aus kleinen Chunk-Größen (z. B. 512 × 512 Pixel) und hochauflösenden DOPs ineffizient ist und zu einem hohen Overhead führte. Dies ist eine über diese Arbeit hinaus wertvolle Erkenntnis, da solche hochauflösenden Daten in bisherigen Workflows weniger berücksichtigt wurden. Dabei zeigt sich auch, dass COG Stärken in der Interoperabilität mit GIS aufzeigt, jedoch weniger optimiert für solche wissenschaftlichen Workflows ist, da essentielle Funktionen wie der partielle Zugriff über S3fs nicht zuverlässig unterstützt werden. Trotzdem zeigt sich, dass solche wissenschaftlichen Workflows mit Xarray auch im Bereich der Geoinformation großes Potenzial bieten. Die Arbeit mit beschrifteten Arrays ist nicht nur benutzerfreundlich, sondern vermeidet auch die Komplexität von reinen NumPy-Arrays. Die Interoperabilität von Xarray mit etablierten Geodatenverarbeitungstools und -formaten stellt daher eine Bereicherung dar.

Die verwendete Architektur zeigt somit einen neuen, spannenden und innovativen Weg der Datenverarbeitung auf, der noch Herausforderungen mit sich bringt, aber ein großes Potential hat und zeigt, was möglich ist, wenn viele Technologien ineinander greifen. Insgesamt zeigt die Arbeit eindrücklich, dass die Verarbeitung von Big Data neue Ansätze erfordert, die über herkömmliche Workflows hinausgehen. Tools wie Dask bieten hier bereits hilfreiche Mechanismen, die jedoch eine tiefere Auseinandersetzung mit den zugrunde liegenden Prozessen erfordern. Es werden zentrale Schwächen und Herausforderungen sichtbar, die vor allem durch die Komplexität der Technologien und die mangelnde Optimierung für hochauflösende Rasterdaten bedingt sind. Dennoch bietet die Architektur eine solide Grundlage, um zukünftige Workflows zu optimieren und die Effizienz raumzeitlicher Analysen weiter zu steigern. Sie demonstriert, dass interdisziplinäre Ansätze und die Zusammenarbeit verschiedener Communities der Schlüssel zur erfolgreichen Arbeit mit Big Data sind.

8.2 Bewertung der Datengrundlage

Die DOPs des LGLN sind eine äußerst wertvolle Datenquelle für eine Vielzahl geowissenschaftlicher Analysen. Diese Arbeit liefert interessante Erkenntnisse zur Nutzung dieser hochauflösenden Daten in Big-Data-Workflows und verdeutlicht zugleich die Potenziale und Optimierungsmöglichkeiten der aktuellen Bereitstellungsform. Die DOPs sind als ARD verfügbar, was einen einfachen Zugang zu den Daten ermöglicht. Besonders auf kleinräumiger Ebene lassen sich daraus relativ einfach Data Cubes erstellen, die raumzeitliche Analysen ermöglichen. Ihre hohe Auflösung und landesweite Abdeckung sind ein großer Vorteil, der jedoch auch spezifische technische Herausforderungen mit sich bringt – insbesondere im Kontext von Big Data.

Dabei zeigt sich, dass hochauflösende Daten bisher wenig in solchen Workflows genutzt werden und durchaus wichtige Erkenntnisse hervorbringen. Ein Beispiel ist die Standard-Kachelgröße von COG mit 512 × 512 Pixeln, die sich in diesem Workflow als suboptimal erwiesen hat. Diese Größe führt zu einem erheblichen Task-Overhead durch eine Vielzahl kleiner Datenfragmente. Besonders bei Kachelgrößen unter einem MB kam es zu massiven Problemen in der Verarbeitung. Größere Kachelgrößen, etwa 1.024 × 1.024 Pixel oder mehr, könnten die Performance in solchen Workflows deutlich verbessern. Zukünftige Untersuchungen sollten klären, wie sich unterschiedliche Kachelgrößen auf andere Anwendungen, wie browserbasierte Tools oder GIS-Workflows, auswirken. Eine optimierte Partitionsgröße, die besser auf die Datenstruktur abgestimmt ist, kann zudem die Dateigröße verringern und die Verarbeitungseffizienz steigern. Besonders effizient erwiesen sich hierbei Partitionsgrößen von 400 × 400 und 2.000 × 2.000

Pixeln (vgl. Kapitel 7.1.1). Die Wahl der optimalen Partitionsgröße hängt somit von der Datenstruktur, der Datenqualität und der räumlichen Auflösung sowie dem jeweiligen Workflow ab. Dies verdeutlicht, dass allgemeingültige Empfehlungen für Aspekte wie Partitions- oder Dateigrößen schwer zu formulieren sind. Vielmehr erfordert jeder Anwendungsfall eine individuelle Betrachtung.

Auch die Wahl der Kompressionsmethode erfordert eine Balance zwischen Speicherbedarf, Performance und verwaltungsrechtlichen Vorgaben. Zwar könnte die Nutzung moderner Kompressionsmethoden wie ZSTD die Dateigröße reduzieren und den Speicherbedarf minimieren, jedoch widerspricht dies den aktuellen Vorgaben der AdV. Da bereits eine verlustbehaftete Kompressionsmethode angeboten wird, muss abgewogen werden, ob diese Speicherreduktion und Kostensenkung sinnvoll erscheint. Die geringe Anzahl verfügbarer Zeitpunkte schränkte die Untersuchung hinsichtlich großräumiger zeitlicher Analysen und der zeitlichen Partitionierung ein. Eine größere Anzahl an Zeitpunkten wäre zukünftig sinnvoll, um weitergehende Erkenntnisse über die Eignung der Formate für raumzeitliche Analysen zu gewinnen.

Zusätzlich hat diese Untersuchung mehrere Optimierungspotenziale für die Bereitstellungsform der DOPs identifiziert, die die Nutzbarkeit im Hinblick auf raumzeitliche Analysen weiter verbessern können. Aktuell sind Zeitstempel nur in den Dateinamen enthalten, was nach dem Herunterladen oder Zusammenführen von Dateien zu Problemen führen kann. Die Integration in die Metadaten würde zeitbezogene Analysen und die Verarbeitung mit Tools wie Xarray erleichtern. Außerdem sollte die fehlerhafte Speicherung des Infrarotkanals als Alphakanal in einigen Jahrgängen korrigiert werden, um Verarbeitungsfehler bei Verwendung der Daten zu vermeiden. Weiterhin treten im großräumigen Kontext spezifische Herausforderungen zutage, die die Komplexität der Daten widerspiegeln. Herausforderungen wie Datenlücken, Überlappungsbereiche oder inkonsistente Zeitstempel durch den Befliegungsturnus erfordern eine umfangreiche Vorverarbeitung für einen konsistenten Data Cube für ganz Niedersachsen. Zur Erleichterung der Data-Cube-Erstellung könnten Anpassungen im STAC-Katalog vorgenommen werden. Eine Möglichkeit wäre, die STAC-Spezifikation so zu modifizieren, dass Bänder als separate Assets definiert werden. Derzeit liegen die Daten in den Assets als gebündelte COG-Dateien vor, was verhindert, dass Bibliotheken wie odc.stac oder stacstac direkt einen Data Cube aus den Items generieren können. Alternativ könnte auch eine spezifische STAC-Erweiterung entwickelt werden, die diesen Prozess erleichtert.

Trotz dieser Herausforderungen verdeutlichen die DOPs des LGLN, dass der Begriff ARD nicht ausschließlich auf Satellitendaten beschränkt bleiben muss. Sie zeigen, dass hochauflösende DOPs einen wertvollen Beitrag zu raumzeitlichen Analysen leisten können. Mit gezielten Anpassungen könnten die DOPs in der zeitlichen Dimension deutlich besser nutzbar gemacht werden, etwa durch die Integration von Zeitstempeln in die Metadaten. Ergänzend dazu könnten technische Verbesserungen wie die Größe des räumlichen Ausschnittes der Datei, die Wahl der Kachelgrößen und eine erleichterte Erstellung von Data Cubes durch Anpassungen im STAC-Katalog oder eine spezifische Erweiterung die Nutzung und Effizienz der Daten weiter steigern. Dies unterstreicht sowohl das Potenzial der Daten als auch die Notwendigkeit einer erweiterten Definition von ARD, die über den traditionellen Fokus auf Satellitendaten hinausgeht.

8.3 Bewertung der Datenformate

Die Formate COG und Zarr repräsentieren zwei unterschiedliche Ansätze zur Speicherung und Analyse raumzeitlicher Rasterdaten. Beide Formate weisen spezifische Stärken und Schwächen auf, die ihre Eignung für unterschiedliche Anwendungsbereiche bestimmen. Die wichtigsten Gemeinsamkeiten und Unterschiede sind in Abbildung 41 zusammengefasst. Diese Visualisierung bietet eine Grundlage für die folgende Bewertung anhand der in Kapitel 4.2 definierten Kriterien.

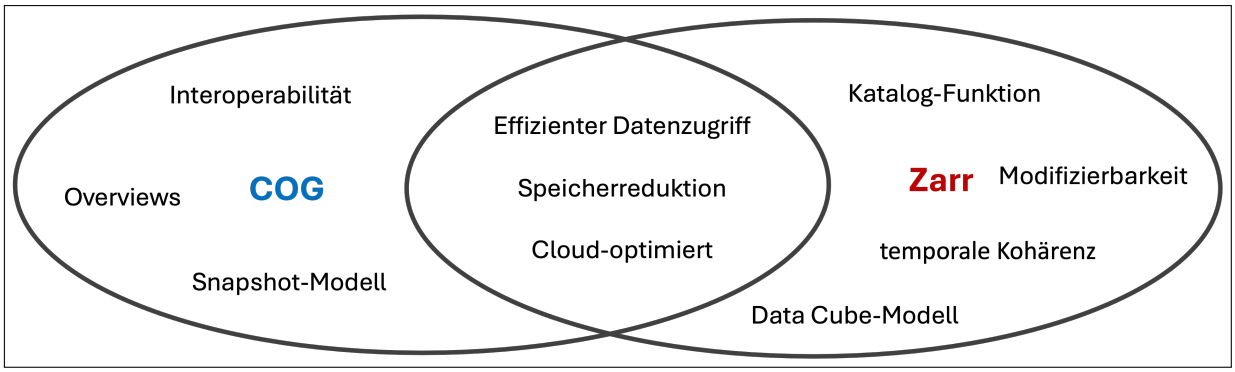

Abbildung 41: Kerneigenschaften von COG und Zarr zur Speicherung und Analyse raumzeitlicher Rasterdaten

Partitionierung und partieller Zugriff:
Wie in Abbildung 41 dargestellt, ermöglichen beide Formate effiziente partielle Zugriffe und parallele Lesevorgänge, die für raumzeitliche Analysen essenziell sind. Die Wahl der Partitionierungsstrategie beeinflusst dabei maßgeblich die Performance. Während kleinere Chunks präzisere Zugriffe erlauben, erhöhen sie die Anzahl an HTTP-Requests, verursachen Overheads und verlängern dadurch die Zugriffszeiten. Partitionen mit mindestens einem MB erwiesen sich in der Analyse als sinnvoll. Die optimale Kachelgröße ist nicht einfach zu bestimmen und variiert je nach Datenstruktur und -größe. In dieser Untersuchung erwies sich eine mittlere Chunk-Größe von etwa 2.000 Pixeln als optimales Gleichgewicht zwischen Effizienz und Performance. Beide Formate unterstützen die parallele Verarbeitung, wobei sich Zarr in dieser Entwicklungsumgebung durch seinen geringeren Implementierungsaufwand und eine höhere Benutzerfreundlichkeit auszeichnete, insbesondere dank der engen Integration mit Xarray und Dask.

Speicherreduktion:
Die Ergebnisse zeigen, dass die Kompression nicht nur den Speicherbedarf erheblich reduziert, sondern auch die Zugriffszeiten optimiert. Beide Formate erfüllen damit die Anforderung einer effizienten Speicherreduktion (siehe Abbildung 41). Dennoch weisen beide Formate bei den verwendeten DOPs moderate Kompressionsraten auf. COG erzielt insgesamt etwas höhere Kompressionsraten als Zarr, insbesondere bei Verwendung der ZSTD- und Deflate-Algorithmen. Aufgrund der Data-Cube-Datenstruktur enthalten Zarr-Dateien potenziell mehr Nullwerte und können bei heterogenen Datenmengen größere Dateigrößen aufweisen. Dieses Problem lässt sich jedoch durch den Ausschluss leerer Chunks effektiv lösen. COG-Dateien eliminieren Nullwerte zwar nicht vollständig, nutzen jedoch eine effiziente Markierung. Durch das Snapshot-Modell ist zudem eine bessere Separierung der Daten möglich, die kleinere, homogenere Datenmengen enthält, was letztlich zu einer effizienteren Speichernutzung führt. Eine zentrale Frage bei der Verwendung von COG ist die optimale Größe der Datei. Diese Frage stellt sich besonders im Kontext von ARD, da die Organisation und Handhabung der Daten direkt mit der Performance zusammenhängt. Größere Dateien bedeuten weniger HTTP-Requests und somit eine effizientere Übertragung und Abfrage der Daten. Gleichzeitig führt das Zusammenfügen von Daten zu einer steigenden Dateigröße, was wiederum die Handhabung erschweren kann. Sehr große Dateien übersteigen schnell die Speicherkapazitäten lokaler Systeme oder erschweren die Verarbeitung in Standard-GIS. Die Frage nach der optimalen Größe ist daher schwer zu beantworten und hängt stark mit dem jeweiligen Anwendungsfall zusammen.

Skalierbarkeit:
Die Größe des Data Cubes hatte keinen signifikanten Einfluss auf die Zugriffszeiten bei ähnlichen Abfragebereichen, was darauf hindeutet, dass Zarr gut skaliert und effizient mit unterschiedlichen Datenvolumina umgehen kann. Zarrs Fähigkeit, Daten ohne Neuorganisation zu erweitern, erhöht die Effizienz bei wachsenden Datenmengen und macht es insbesondere für mehrdimensionale Daten deutlich skalierbarer als COG. COG-Dateien können zwar ebenfalls in skalierbaren Umgebungen erfolgreich verarbeitet werden und bieten durch Tiling gute Möglichkeiten, stoßen jedoch bei der Erweiterung um neue Daten oder Dimensionen an Grenzen, da Änderungen oft ein vollständiges Neuschreiben der Dateien erfordern.

Modifizierbarkeit:
Zarr ermöglicht flexible Datenmodifikationen, einschließlich des Hinzufügens neuer Zeitpunkte oder räumlicher Ausschnitte (siehe Abbildung 41). Änderungen können gezielt auf Chunk-Ebene erfolgen, ohne die gesamte Datei neu schreiben zu müssen, was die Effizienz steigert. Parallele Schreibvorgänge in Zarr erfordern jedoch eine sorgfältige Handhabung zur Sicherstellung der Datenintegrität, und auch die Transaktionssicherheit muss gewährleistet sein, um eine robuste Verarbeitung zu ermöglichen. COG hingegen ist in diesem Bereich limitiert, da jede Änderung ein vollständiges Neuschreiben der Datei erfordert. Dies macht COG für dynamische Workflows weniger geeignet, gewährleistet aber eine höhere Robustheit hinsichtlich der Datenintegrität.

Raumzeitliche Datenstrukturen:
Zarr ermöglicht eine effizientere Verarbeitung raumzeitlicher Daten durch die native Unterstützung mehrdimensionaler Strukturen. Das Data-Cube-Modell von Zarr bietet temporale Kohärenz und macht das Format besonders interessant für Langzeituntersuchungen und Trendanalysen in Big-Data-Szenarien (siehe Abbildung 41). Allerdings erfordert dies eine sorgfältige Handhabung, um die Robustheit zu gewährleisten. Insbesondere die Dokumentation von Zarr bietet im Kontext geodatenbezogener Anwendungen Verbesserungspotenzial. Dank der aktiven Community konnten jedoch viele Workarounds aus Foren und Webinaren identifiziert werden. CRS-Informationen oder andere geodatenspezifische Metadaten müssen vom Ersteller explizit und sorgfältig definiert werden, was zu potenziellen Fehlerquellen führen kann. Während Zarr durch seine flexible Struktur überzeugt, setzt die Nutzung ein tieferes Verständnis der Datenstruktur voraus, was die Benutzerfreundlichkeit einschränkt. COG punktet hier mit einer robusten und intuitiven Handhabung sowie einer guten Dokumentation der Verarbeitungsbibliotheken, ist jedoch weniger flexibel für mehrdimensionale Strukturen. Die integrierte Katalogfunktion von Zarr erleichtert das Datenmanagement, erfordert jedoch eine saubere Strukturierung durch die Erstellenden (siehe Abbildung 41). Die Datenintegrität von Zarr ist geringer, da die Datenkonsistenz in der Verantwortung der Nutzenden und der erstellenden Instanz liegt. Fehlerhafte oder unstrukturierte Data Cubes erschweren die Nutzung und setzen voraus, dass die Ersteller ein tiefes Verständnis für die Datenstruktur haben. Dies macht die Nutzung komplexer und weniger intuitiv, weshalb viele Data Cubes individuell erstellt werden. Bei COG liegt die Verantwortung für Konsistenz oft bei externen Verwaltungsinstanzen.

Standardisierung und Interoperabilität:
Zarr ist im Bereich der Geoinformation bislang weniger interoperabel als COG. In dieser Entwicklungsumgebung zeigt sich jedoch, dass Zarr in Bezug auf Effizienz und Skalierbarkeit durch die enge Integration mit Xarray und Dask deutlich optimiert ist. Dies spiegelt sich insbesondere im geringeren Implementierungsaufwand und der verbesserten Benutzerfreundlichkeit wider. Dies unterstreicht das Streben nach cloud-native Geospatial-Lösungen und verdeutlicht, dass Benutzerfreundlichkeit nicht nur vom Format selbst ausgeht. COG basiert auf dem weit verbreiteten GeoTIFF-Standard und ist mit den meisten GIS-Plattformen und -Tools wie GDAL, Rasterio oder Rio-Cogeo kompatibel, was die Integration und den Datenaustausch erleichtert und auf der linken Seite von Abbildung 41 betont wird. Die Standardisierung von COG erhöht die Robustheit im Datenaustausch und der Interoperabilität und bietet eine hohe Benutzerfreundlichkeit. Die Dateien können leicht heruntergeladen, lokal verarbeitet und in kleinen Abschnitten bearbeitet werden. Dies macht ARD-basierte Workflows zugänglich für eine breite Nutzerbasis. Auch für „on the fly“-Bearbeitung, wo mehrdimensionale Strukturen zwischenzeitlich erzeugt werden, aber nicht gespeichert werden müssen, bietet sich COG an. Die Entwicklung des Zarr-Formats zeigt sich in der wachsenden Nutzung. Sollte sich dieser Trend fortsetzen, werden die Interoperabilität und Stabilität des Zarr-Ökosystems von entscheidender Bedeutung sein, um das volle Potenzial dieser Datensätze auszuschöpfen. Gleichzeitig bleibt die groß angelegte Speicherung von Array-Daten ein aktiver Bereich technologischer Innovation. Insbesondere die Nutzung von Zarr als Data Lake ist interessant, da dabei nicht nur die Bereitstellung, sondern auch die dynamische Veränderung von Daten adressiert wird. Diese Veränderlichkeit setzt jedoch die Integration zusätzlicher Softwarelösungen wie Arraylake voraus, um

Datenkonsistenz, Transaktionssicherheit und Nutzerrollen zu gewährleisten. Diese Ausrichtung macht Zarr zu einer zukunftsweisenden Lösung, die jedoch Zeit benötigt, um sich vollständig zu etablieren.

Übersichten:
COG bietet durch native Unterstützung von räumlichen Overviews erhebliche Vorteile bei der Datenvisualisierung und Analyse in verschiedenen Zoomstufen (siehe Abbildung 41). Zarr ermöglicht zwar zeitliche und räumliche Aggregationen, diese müssen jedoch manuell erstellt werden. Dadurch ist Zarr flexibler, aber aufwendiger in der Handhabung.

Zusammenfassung:
Die Analyse zeigt, dass beide Formate ihre Stärken und Schwächen haben. Zarr ist besonders effizient und skalierbar bei der Verarbeitung großer, multidimensionaler Datensätze und eignet sich für dynamische Workflows sowie komplexe raumzeitliche Analysen. Es bietet hohe Flexibilität, erfordert jedoch spezialisierte Kenntnisse und Werkzeuge, was die Benutzerfreundlichkeit und Robustheit beeinträchtigen kann. COG punktet mit hoher Benutzerfreundlichkeit, Standardisierung und breiter Unterstützung in GIS-Anwendungen. Es ist vertraut und einfach zu nutzen, ideal für lokale und kleinere Analysen mit geringem Änderungsbedarf. Allerdings ist COG weniger flexibel bei mehrdimensionalen Datenstrukturen und dynamischen Anforderungen. Die Bewertung dieser Aspekte wird in Abbildung 42 grafisch zusammengefasst. Es sei darauf hingewiesen, dass diese Einschätzung auf subjektiven Erfahrungen während der Arbeit mit den Formaten basiert und sich auf die spezifische Architektur dieser Arbeit (vgl. Kapitel 4.5) bezieht. Die Ergebnisse sind daher in ihrer Aussagekraft begrenzt und sollen als Orientierung für weiterführende Untersuchungen dienen. Eine weiterführende Betrachtung ist insbesondere in Bezug auf tiefere Einblicke mit einer umfassenderen zeitlichen Dimension, alternativen Workflows und Architekturen sowie unterschiedlichen Datengrundlagen von großem Interesse. Zukünftige Untersuchungen könnten auch zusätzliche Messgrößen wie die Anzahl der HTTP-Requests, die übertragene Datenmenge sowie erweiterte raumzeitliche Analysen und die Nutzung von Übersichten einbeziehen, um weitere Erkenntnisse zu gewinnen.

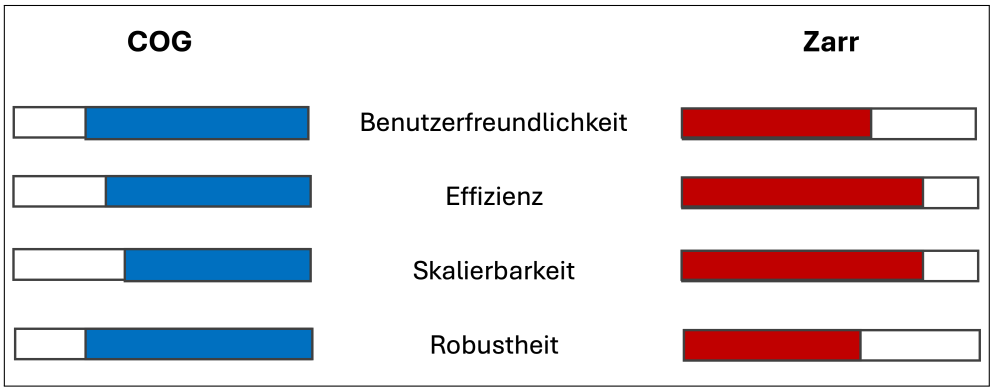

Abbildung 42: Vergleich von COG und Zarr zur Speicherung und Analyse raumzeitlicher Rasterdaten

Die Wahl des passenden Formats erfordert einen Kompromiss zwischen Benutzerfreundlichkeit, Effizienz und Flexibilität. Beide Formate zeigen, dass technologische Innovationen neue Möglichkeiten und Herausforderungen schaffen. Entscheidend ist eine sorgfältige Abwägung der Anforderungen, wie Datenmenge, Analyseart und Änderbarkeitsanforderungen, um das optimale Format für den jeweiligen Kontext zu bestimmen.

COG eignet sich besonders für lokale, kleinere Analysen im GIS-Bereich, bei denen wenige Datenänderungen erforderlich sind. Kleinere Dateigrößen können hier vorteilhaft sein, da weniger Dateien umgeschrieben werden müssen. Zarr hingegen zeigt seine Stärken bei großen Datenmengen, mehrdimensionalen Strukturen und Analysen, die räumliche und zeitliche Dimensionen kombinieren, und bietet dabei hohe Effizienz. Die Zeitdimension spielt bei der Wahl zwischen COG und Zarr eine entscheidende

Rolle. Für zeitpunktbasierte Analysen, wie die Berechnung von NDVI-Statistiken oder andere Bild-für-Bild-Auswertungen, sind beide Formate geeignet. COG ermöglicht es, jedes Bild einzeln zu verarbeiten und die Ergebnisse anschließend zusammenzuführen. Für Analysen, die zeitintervallgestützte Auswertungen erfordern wie z. B. Trendanalysen, Histogramme oder Interpolationen bietet Zarr entscheidende Vorteile. Solche Anwendungen erfordern häufig das Zusammenfügen von Daten in der räumlichen und/oder der zeitlichen Dimension, was mit COG aufwendig und datenintensiv ist. Zarr zeigt durch die integrative Datenstruktur Vorteile, insbesondere bei einer hohen Anzahl an Zeitpunkten.

Zudem hängt die Wahl des Formates auch von dem Workflow ab. Ein konkretes Beispiel ist die Verarbeitung mehrdimensionaler Daten in der KI-basierten Gebäudeerkennung. Hierbei kombiniert das LGLN mehrere Datenquellen wie DOPs (im COG-Format) und bildbasierte, digitale Oberflächenmodelle (bDOM) zu einem mehrdimensionalen Array mit Höheninformationen. Dieses Array dient als Eingabedatensatz für ein KI-Modell. Im aktuellen Workflow wird das Array nur einmalig und „on the fly“ verarbeitet, wodurch eine persistente Speicherung nicht erforderlich ist. Sollten diese Daten jedoch wiederholt verwendet werden müssen oder als Trainingsdatensätze für KI-Modelle bereitgestellt werden, ist eine Speicherung sinnvoll. In diesem Fall wäre Zarr eine gute Wahl, da es ermöglicht, neue Daten flexibel hinzuzufügen und bestehende Arrays zu aktualisieren. Damit ließen sich iterative Trainingsprozesse oder die Bereitstellung von KI-Labels deutlich vereinfachen. Durch die Kombination von Zarr mit Werkzeugen wie IceChunk könnten zusätzlich Versionierungen und eine erhöhte Transaktionssicherheit integriert werden, was die Anwendung dieses Formats in einem solchen Workaround sehr interessant macht. Die Analyse der Formate COG und Zarr zeigt, dass es kein absolutes Datenformat gibt, das allen Anforderungen gerecht wird. Die Wahl des Formats hängt letztendlich von den spezifischen Anforderungen und Zielen ab. Bevor ein Datenformat ausgewählt wird, sollten zentrale Fragen geklärt werden:

1. **Anwendungsumgebung:**
    - Sollen die Daten lokal oder in der Cloud verarbeitet werden?
    - Ist ein direkter Online-Zugriff erforderlich oder werden die Daten heruntergeladen?
2. **Zeitliche Dimension**:
    - Sind zeitpunktbasierte Berechnungen (Statistiken, Indizes) oder zeitintervallgestützte Analysen (Trends, Histogramme, Interpolationen) erforderlich?
    - Wie umfangreich sind die betrachteten Zeitreihen?
3. **Verarbeitungsworkflow:**
    - Wird eine einmalige Verarbeitung (on the fly) oder ein wiederholter Workflow benötigt?
    - Werden kleine Datenmengen schrittweise oder große Mengen auf einmal verarbeitet?
4. **Modifizierbarkeit:**
    - Wie häufig müssen die Daten aktualisiert oder erweitert werden?

9 Fazit

Insgesamt zeigt sich, dass die cloud-optimierten Formate ein wichtiger erster Schritt sind, um große Datenmengen zugänglicher zu machen. Beide Formate zeigen Stärken, insbesondere durch die partielle Zugriffsmöglichkeit und Speicherreduktion. Dennoch bleibt dies nur ein erster Schritt. Um die Bereitstellung und Nutzung großer raumzeitlicher Rasterdaten wirklich zu erleichtern, braucht es nicht nur leistungsfähige Formate, sondern auch geeignete Tools, Software und Plattformen, die einfache Suche, Auffindbarkeit und Verwendbarkeit sicherstellen. Ansätze wie STAC in Kombination mit COG und die Projekte der Zarr-Community, etwa Pangeo oder Arraylake, bieten hier vielversprechende Lösungen.

Zarr zeichnet sich durch seine interaktive Arbeitsweise aus, die flexible und effiziente Workflows ermöglicht. Insgesamt ist Zarr dabei ein spannender Ansatz, der datenbank-, datei- und arraybasierte Herangehensweisen vereint und viele Vorteile mit sich bringt, gerade was das Lesen und Schreiben von Dateien angeht. Die Struktur von Zarr ist aufgrund der binären Dateien simpel, verständlich und effektiv. Daten können während der parallelen und verteilten Verarbeitung direkt ergänzt, modifiziert und manipuliert werden. Dies ermöglicht flexible und effiziente Workflows, die für Big-Data-Analysen entscheidend sind. Die bestehende Integration mit Verarbeitungsbibliotheken wie Xarray und Dask zeigt zudem, wie wichtig interdisziplinäre Ansätze und die Zusammenarbeit verschiedener Communities sind, um die Herausforderungen im Umgang mit großen Datenmengen erfolgreich zu bewältigen. Tools wie Arraylake oder IceChunk erleichtern zusätzlich die Versionierung und Transaktionssicherheit, wodurch Zarr für Big-Data-Szenarien zukunftsweisend ist. Für typische GIS-Analysen bleibt COG die bessere Wahl, da es weniger technische Komplexität erfordert und eine breitere Interoperabilität bietet. Der Ansatz mit COG und STAC bietet Lösungen für ein breites Anwendunsgspektrum. Auch viele zeitliche Analysen sind mit einem gewissen Workaround gut mit COG machbar. COG überzeugt im Vergleich zur Data Cube-Struktur durch einfache Handhabung, Interoperabilität und Eignung für GIS-basierte und punktuelle Analysen, jedoch auf Kosten der Performance. Zarr zeigt sich dabei insgesamt performanter, insbesondere bei steigender Datenmenge.

Die Untersuchung mit den DOPs vom LGLN zeigt jedoch, wie komplex es ist, zugriffsbereite Daten zur Verfügung zu stellen. Hohe Auflösungen erfordern gut durchdachte Workflows, etwa bei der Wahl geeigneter Chunk- und Dateigrößen.Während auf lokaler Ebene die Zusammenstellung eines Data Cubes relativ einfach gelingt, fehlen den DOPs in Niedersachsen aktuell die Voraussetzungen für eine konsistente ARD-Struktur. Unterschiedliche Befliegungszyklen und Datenlücken erschweren die Bildung großflächiger, zeitlich harmonisierter Data Cubes und unterstreichen die Herausforderungen bei der Integration der Zeitdimension.

Die Arbeit zeigt, dass es kein universell bestes Format gibt, sondern dass die Entscheidung für ein Format ein Abwägen zwischen Benutzerfreundlichkeit, Flexibilität und Performance darstellt. Die Wahl des Formats sollte daher sorgfältig anhand von Faktoren wie Anwendungsumgebung, zeitlicher Dimension, Verarbeitungsworkflow und Modifizierbarkeit der Daten getroffen werden. Beide Formate leisten einen wichtigen Beitrag zur Speicherung und Analyse großer raumzeitlicher Rasterdaten und sind integrale Bestandteile moderner Geoinformationssysteme. Gleichzeitig zeigt sich, dass Big-Data-Szenarien oft abweichende Workflows erfordern, die ein Umdenken und neue Herangehensweisen in der Datenverarbeitung nötig machen. Die entwickelte Architektur auf Basis von Kubernetes, Dask und Xarray bietet eine flexible und leistungsstarke Grundlage für die Verarbeitung großer Datenmengen. Sie verdeutlicht, dass eine ausgereifte Infrastruktur essenziell ist, um die Potenziale cloud-optimierter Datenformate auszuschöpfen. Es zeigt sich auch, dass dieser Bereich aktuell stark im Trend liegt und intensiv erforscht sowie weiterentwickelt wird. Für zukünftige Entwicklungen wird es entscheidend sein, die Interoperabilität von Formaten wie Zarr weiter auszubauen und gleichzeitig bestehende Standards der OGC, AdV und anderen Stellen, sowie nicht zuletzt COG selbst für multidimensionale Daten anzupassen. Durch die kontinuierliche Weiterentwicklung cloud-optimierter Datenformate und deren Integration in bestehende Workflows können die Herausforderungen der Big-Data-Verarbeitung im Geodatenmanagement bewältigt und neue Potenziale erschlossen werden.

10 Ausblick

Die Ergebnisse dieser Arbeit zeigen, dass das Management großer raumzeitlicher Rasterdaten ein dynamisches und sich stetig weiterentwickelndes Forschungsfeld ist. Cloud-optimierte Datenformate wie COG und Zarr bieten vielversprechende Ansätze, um die Herausforderungen bei der Speicherung und Analyse dieser Datenmengen zu bewältigen. Gleichzeitig werfen sie jedoch zahlreiche Forschungsfragen auf, die technische, organisatorische und anwendungsorientierte Aspekte betreffen. Eine der zentralen Herausforderungen bei der Nutzung von COG ist die Frage nach optimalen Kachel- und Dateigrößen.

Diese Parameter beeinflussen maßgeblich die Zugriffsgeschwindigkeit, den Speicherbedarf und die Interoperabilität. Bislang fehlen standardisierte Best Practices, die sich auf unterschiedliche Anwendungsfälle übertragen lassen. Zudem bleibt offen, wie COG für zeitliche Analysen und mehrdimensionale Daten erweitert werden könnte, um eine flexiblere Nutzung zu ermöglichen. Für Zarr hingegen liegt der Fokus aktuell auf der Verbesserung der Interoperabilität, insbesondere im GIS-Bereich sowie auf der Erforschung neuer Einsatzmöglichkeiten wie dynamischer Data Lakes.

Eine übergeordnete Fragestellung ist, wie viele Nutzende tatsächlich regelmäßig mit derart großen Datenmengen arbeiten. Es bleibt zu untersuchen, ob die Entwicklung von Technologien wie Zarr und COG vor allem einem spezialisierten Nutzerkreis dient oder ob diese Formate langfristig einen breiteren Nutzen bieten können. Im Zusammenhang damit ist auch die Bereitstellung vorgefertigter Data Cubes kritisch zu hinterfragen. Einerseits beschleunigen vorgefertigte Data Cubes die Analyseprozesse, andererseits könnten sie den Nutzerkreis durch spezifische Anforderungen einschränken. Flexible Werkzeuge, mit denen Nutzende selbst Data Cubes erstellen können, könnten hier eine bessere Lösung sein.

Auf infrastruktureller Ebene zeigt die Arbeit ebenfalls Optimierungsbedarf. Während Dask als Werkzeug für parallele und verteilte Verarbeitung flexibel ist, bringt es auch eine gewisse Unvorhersehbarkeit mit sich. Alternativen wie Cubed könnten hier getestet werden, um stabilere und besser planbare Workflows zu ermöglichen. Darüber hinaus wäre es interessant zu untersuchen, wie sich unterschiedliche Cloud-Plattformen (z. B. AWS, Azure oder Google Cloud) hinsichtlich Performance, Skalierbarkeit und Kosten auswirken und welche Systeme sich am besten für das Management raumzeitlicher Rasterdaten eignen.

Aus Sicht des LGLN könnte als erster Schritt eine Evaluation der eingesetzten Technologie mit COG und STAC erfolgen, um zu prüfen, welche Kachelgrößen und Dateigrößen sich am besten für interne und externe Anwendungsfälle eignen. Zusätzlich sollte untersucht werden, inwieweit Anpassungen für raumzeitliche Abfragen wie die Erweiterung um zeitliche Metadaten oder die Erleichterung der Erstellung von Data Cubes aus COG-Daten vorgenommen werden können. Daraus ergibt sich auch die Fragestellung für welche Analysen und Services raumzeitliche Rasterdaten denkbar sind wie beispielsweise zur Überwachung von Umweltveränderungen im Umweltservices-Team und ob dabei auch Big-Data-Analysen für ganz Niedersachsen relevant sind. Dabei stellt sich als Geobasisanbieter die Frage, ob die Bereitstellung vorgefertigter Data Cubes sinnvoll ist oder ob flexible Tools, mit denen Anwendende spezifische Data Cubes selbst erstellen können, besser geeignet wären. Weiterhin könnte geprüft werden, ob Zarr in Kombination mit Tools wie Arraylake oder IceChunk genutzt werden kann, um einen Data Lake für Raster- und multidimensionale Rasterdaten zu schaffen. Ein solcher Data Lake könnte nicht nur eine effiziente Speicherung großer Datenmengen ermöglichen, sondern auch dynamische Analysen und flexible Datenmodifikationen unterstützen. Dabei wäre es sinnvoll, die technischen Anforderungen und den Nutzen eines Zarr-basierten Data Lakes mit Blick auf den Workflow des LGLN zu untersuchen.

Das Thema der Speicherung und Analyse großer raumzeitlicher Rasterdaten ist insgesamt noch neu und befindet sich in einem dynamischen Wandel. Technologische Fortschritte und innovative Ansätze werden kontinuierlich entwickelt, sodass es spannend bleibt zu beobachten, welche Entwicklungen die Zukunft bringt, welche Technologien sich durchsetzen und wie Institutionen wie das LGLN ihre Datenangebote anpassen, um den Bedürfnissen einer breiten Nutzerschaft gerecht zu werden.

Literaturverzeichnis

Abdishakur, H. (2022). Zarr - Cloud Natives Geospatial Data Format. *Spatial Data Science*. https://medium.com/spatial-data-science/zarr-cloud-native-geospatial-data-format-b6cc445e625 (Abgerufen am: 09.04.2024).

Abernathey, R. (2024). Announcing icechunk! https://earthmover.io/blog/icechunk (Abgerufen am: 10.11.2024).

Abernathey, R. P., Augspurger, T., Banihirwe, A., Blackmon-Luca, C. C., Crone, T. J., Gentemann, C. L., Hamman, J. J., Henderson, N., Lepore, C., McCaie, T. A., Robinson, N. H., & Signell, R. P. (2021). Cloud-Native Repositories for Big Scientific Data. *Computing in Science and Engineering*, *23*(2), 26–35.

Alam, M. M., Torgo, L., & Bifet, A. (2022). A Survey on Spatio-temporalData Analytics Systems. *ACM ComputingSurveys*, *54*(10), 1–38.

Alberti, K. (2018). Guide to GeoTIFF compression and optimization with GDAL. https://kokoalberti.com/articles/geotiff-compression-optimization-guide/ Abgerufen am 03.11.2024.

Anaconda, I., & Contributors. (2016). Managing Computation. https://distributed.dask.org/en/stable/manage-computation.html (Abgerufen am 17.10.2024).

Anaconda, Inc. and Contributors. (2018a). 10 Minutes to Dask. https://docs.dask.org/en/stable/10-minutes-to-dask.html (Abgerufen am: 13.08.2024).

Anaconda, Inc. and Contributors. (2018b). Bag. https://docs.dask.org/en/stable/bag.html (Abgerufen am: 13.08.2024).

Anaconda, Inc. and Contributors. (2018c). Dask Best Practices. https://docs.dask.org/en/stable/best-practices.html (Abgerufen am: 10.08.2024).

Anaconda, Inc. and Contributors. (2018d). Deploy Dask Clusters. https://docs.dask.org/en/stable/deploying.html (Abgerufen am: 19.09.2024).

Bengel, G., Baun, C., Kunze, M., & Stucky, K.-U. (2015). *Masterkurs Parallele und Verteilte Systeme: Grundlagen und Programmierung von Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und Cloud* (2. Aufl.). Springer Vieweg Wiesbaden.

Bill, R. (2016). *Grundlagen der Geo-informationssysteme*. Wichmann Berlin.

Blosc Developers. (2024). Blosc: A blocking, shuffling and lossless compression library. GitHub. https://github.com/Blosc/c-blosc (Abgerufen am 06.11.2024).

BMI. (2012). Vorsprung durch Geoinformationen. 3. Bericht der Bundesregierung über die Fortschritte zur Entwicklung der verschiedenen Felder des Geoinformationswesens im nationalen, europäischen und internationalen Kontext. https://www.imagi.de/SharedDocs/downloads/Webs/IMAGI/DE/Geofortschrittsberichte/3-fortschrittsbericht.pdf (Abgerufen am: 16.04.2024).

Bohm, S., & Beranek, J. (2020). Runtime vs Scheduler: Analyzing Dask's Overheads. *IEEE/ACM Workflows in Support of Large-Scale Science*, 1–8. https://doi.ieeecomputersociety.org/10.1109/WORKS51914.2020.00006

Böhm, S., & Beránek, J. (2020). Runtime vs Scheduler: Analyzing Dask's Overheads. *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 1–8. https://doi.org/10.1109/WORKS51914.2020.00006

Buckley, G. (2021). Choosing good chunk sizes in Dask. https://blog.dask.org/2021/11/02/choosing-dask-chunk-sizes (Abgerufen am: 10.08.2024).

Carré, C., & Hamdani, Y. (2021). Pyramidal Framework: Guidance for the Next Generation of GIS Spatial-Temporal Models. *ISPRS International Journal of Geo-Information*, *10*(3), 1–41.

CEOS. (n. d.). CEOS Analysis Ready Data. https://ceos.org/ard/ (Abgerufen am: 14.10.2024).

Chris Holmes et. al. (2024). Cloud Optimized GeoTIFF. https://www.cogeo.org (Abgerufen am: 20.04.2024).

Cloud Native Computing Foundation. (2024). CNCF Cloud Native Definition v1.1. https://github.com/cncf/toc/blob/main/DEFINITION.md (Abgerufen am: 20.08.2024).

Cloud-Native Geospatial Foundation. (2023). Advanced COG/GeoTIFF Details. https://guide.cloudnativegeo.org/cloud-optimized-geotiffs/cogs-details.html (Abgerufen am 03.11.2024).

cogeotiff. (n. d.). rio-cogeo. https://cogeotiff.github.io/rio-cogeo/ Abgerufen am 12.11.2024.

CyberSWIFT. (2022). How Cloud Storage and Geospatial Data Work Together in Advanced Technology? https://cyberswift.medium.com/why-in-advanced-technology-are-geospatial-data-and-cloud-storage-irreplaceable-518e7abde763 (Abgerufen am: 20.04.2024).

Daniel, J. C. (2019). *Data Science with Python and Dask*. Manning Publications.

Daniel O'Donohue. (2023). GeoTIFF Compression. https://mapscaping.com/geotiff-compression-techniques/ (Abgerufen am 06.11.2024).

Dask core developers. (2024). Dask. https://www.dask.org (Abgerufen am: 13.08.2024).

Dask Developers. (2017). Dask-ML. https://ml.dask.org (Abgerufen am: 13.08.2024).

Dask Developers. (2018). Welcome to the Dask Tutorial. https://tutorial.dask.org/00_overview.html (Abgerufen am: 13.08.2024).

Data Science Wizards. (2023). Introduction to Time Series Analysis. https://medium.com/@datasciencewizards/introduction-to-time-series-analysis-i-82d614d462b0 (Abgerufen am: 14.09.2024).

Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR). (n. d.). Das CODE-DE Portfolio. https://code-de.org/de/portfolio/ (Abgerufen am: 06.09.2024).

Di, L., & Sun, Z. (2023). Cloud Computing and Cloud Service. *Encyclopedia of Mathematical Geosciences*, 123–127.

Di, L., & Yu, E. (2023). Big Data Analytics for Remote Sensing: Concepts and Standards. *Remote Sensing Big Data*, 155–170.

Ferreira, K. R., Queiroz, G. R., Vinhas, L., Marujo, R. F. B., Simoes, R. E. O., Picoli, M. C. A., Camara, G., Cartaxo, R., Gomes, V. C. F., Santos, L. A., Sanchez, A. H., Arcanjo, J. S., Fronza, J. G., Noronha, C. A., Costa, R. W., Zaglia, M. C., Zioti, F., Korting, T. S., Soares, A. R., … Fonseca, L. M. G. (2020). Earth Observation Data Cubes for Brazil: Requirements, Methodology and Products. *Remote Sensing*, *12*(24), 1–19.

Freeman, J., Martin, K., & Hamman, J. (2021). A new toolkit for data-driven maps. https://carbonplan.org/blog/maps-library-release (Abgerufen am: 09.04.2024).

FreeWheel Biz-UI Team. (2024). *Cloud-Native Application Architecture : Microservice Development Best Practice* (1. Aufl.). Springer Singapore.

Giuliani, G., Chatenoux, B., Bono, A. D., Rodila, D., Richard, J.-P., Allenbach, K., Dao, H., & Peduzzi, P. (2017). Building an Earth Observations Data Cube: lessons learned from the Swiss Data Cube (SDC) on generating Analysis Ready Data (ARD). *Big Earth Data*, *1*(1-2), 100–117.

Giuliani, G., Chatenoux, B., Piller, T., Moser, F., & Lacroix, P. (2020). Data Cube on Demand (DCoD): Generating an earth observation Data Cube any-where in the world. *International Journal of Applied Earth Observation and Geoinformation*, *87*, 1–6.

Goniwada, S. R. (2022). *Cloud native architecture and design : a handbook for modern day architecture and design with enterprise-grade examples* (1. Aufl.). Apress Berkeley, CA.

Gustafson, J. (1988). Reevaluating Amdahl's Law. *Communications of the ACM*, *31*(5), 532–533.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach* (6. Aufl.). Morgan Kaufmann.

Holmes, C. (2021). Towards a Cloud-Native Geospatial standards baseline. https://www.ogc.org/blog-article/towards-a-cloud-native-geospatial-standards-baseline/ (Abgerufen am: 04.04.2024).

Hoyer, S., & Hamman, J. J. (2017). Xarray: N-D labeled Arrays and Datasets in Python. *Journal of Open Research Software*, *5*, 1–6.

Jetter, F. (2023). Reduce memory pressure for multi array reductions by releasing splitter tasks more eagerly. https://github.com/dask/dask/pull/10535 (Abgerufen am: 12.09.2024).

Ji, C., Li, Y., Qiu, W., Awada, U., & Li, K. (2012). Big Data Processing in Cloud Computing Environments. *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*, 17–23.

Jim Crist-Harif. (2021). Install on a Kubernetes Cluster. https://gateway.dask.org/install-kube.html (Abgerufen am: 09.09.2024).

Joseph, G. (2023). over-prioritizes root tasks in some situations. https://github.com/dask/dask/pull/9995 (Abgerufen am: 12.09.2024).

Kavouras, M. (2001). Understanding and Modelling Spatial Change. *Life and Motion of Socio-Economic Units*, 1–11.

Khushalani, P. (2022). *Kubernetes Application Developer - Develop Microservices and Design a Software Solution on the Cloud* (1. Aufl.). Apress Media.

Kopp, S., Becker, P., Doshi, A., Wright, D. J., Zhang, K., & Xu, H. (2019). Achieving the Full Vision of Earth Observation Data Cubes. *MDPI*, *4*(94), 1–19.

Kristen Evans. (2018). Introducing the Cloud Native Landscape 2.0 – interactive edition. https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/ (Abgerufen am: 20.08.2024).

Kuzmiakova, A. (2022). *Concurrent, Parallel and Distributed Computing*. Arcler Education Incorporated.

Landesamt für Geoinformation und Landesvermessung Niedersachsen. (2021). Digitale Orthophotos (DOP).

Landesamt für Geoinformation und Landesvermessung Niedersachsen. (n. d.). Luftbilder und Digitale Orthophotos des ATKIS (ATKIS-DOP). https://www.lgln.niedersachsen.de/startseite/geodaten_karten/luftbildprodukte/luftbilder-orthophotos-142298.html (Abgerufen am 25.10.2024).

LZ4 developers. (2024). LZ4 - Extremely fast compression algorithm. GitHub. https://github.com/lz4/lz4/tree/dev (Abgerufen am 06.11.2024).

MapScaping Aps. (2023). Rasters In A Database? https://mapscaping.com/podcast/rasters-in-a-database/ (Abgerufen am: 15.09.2024).

Marshall, E. (2024). Vector data cubes in xarray. https://earthmover.io/blog/vector-datacube-pt1 (Abgerufen am 16.11.2024).

Maskey, M., Ramachandran, R., Freitag, B., Kaulfus, A., Barciauskas, A., Veerman, O., Thomas, L., Gurung, I., & Ramasubramanian, M. (2023). Dashboard for Earth Observation (1. Aufl.). *Advances in Scalable and Intelligent Geospatial Analytics: Challenges and Applications*, 203–222.

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing* (Techn. Ber.). National Institute of Standards und Technology. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (Abgerufen am: 10.05.2024).

Miles, A. (2019). Zarr: Scalable Storage of Tensor Data for Use in Parallel and Distributed Computing | SciPy 2019 |. https://www.youtube.com/watch?v=qyJXBlrdzBs (Abgerufen am: 09.04.2024).

Moreno, M., Vilaça, R., & Ferreira, P. (2022). Scalable transcriptomics analysis with Dask: applications in data science and machine learning. *BMC Bioinformatics*, *23*(514), 1–20.

Nicholas Skytland. (2014). What is NASA doing with Big Data? https://skytland.medium.com/what-is-nasa-doing-with-big-data-a6c748588588 (Abgerufen am: 14.10.2024).

Nwaeze, D. (2024). Webinar: Cloud-Native Geospatial - African Perspectives. https://www.youtube.com/watch?v=6bAsgmjCe0Y%20(11:28) (Abgerufen am 17.10.2024).

OGC. (2024). OGC Testbed 19 Analysis Ready Data Engineering Report.

Open Geospatial Consortium. (2022a). OGC Cloud Optimized GeoTIFF standard candidate. https://docs.ogc.org/is/21-026/21-026.html (Abgerufen am: 08.04.2024).

Open Geospatial Consortium. (2022b). Overview: Zarr - A Cloud Native ND Array Format. https://www.youtube.com/watch?v=KiiKvXzhyMs%20(17:11) Abgerufen am 02.11.2024.

Open Geospatial Consortium. (2022c). Zarr Storage Specification 2.0 Community Standard.

Open Geospatial Consortium. (n. d.). About. https://www.ogc.org/about-ogc/ (Abgerufen am: 04.04.2024).

Palach, J. (2014). *Parallel Programming with Python* (1. Aufl.). Packt Publishing.

Parhami, B. (2018). Parallel Processing with Big Data. *Encyclopedia of Big Data Technologies*, 1–7.

Peters, T. (2023). *Parallel Python with Dask: Perform distributed computing, concurrent programming and manage large dataset*. GitforGits.

Petrelli, M. (2023). Parallel Computing and Scaling with Dask. *Machine Learning for Earth Sciences: Using Python to Solve Geological Problems*, 161–175.

Planet Labs PBC. (2024). An Introduction to Cloud Optimized GeoTIFFS (COGs) Part 1: Overview. https://developers.planet.com/docs/planetschool/an-introduction-to-cloud-optimized-geotiffs-cogs-part-1-overview/ (Abgerufen am: 08.04.2024).

Ristov, S., Prodan, R., Gusev, M., & Skala, K. (2016). Superlinear speedup in HPC systems: Why and when? *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 889–898.

Simoes, R., Camara, G., Queiroz, G., Souza, F., Andrade, P. R., Santos, L., Carvalho, A., & Ferreira, K. (2021). Satellite Image Time Series Analysis for Big Earth Observation Data. *Remote Sensing*, *13*(13), 1–20.

Singh, A. (2021). *Parallel And Distributed Computing*.

Sisodiya, N., Dube, N., Prakash, O., & Thakkar, P. (2023). Scalable big earth observation data mining algorithms: a review. *Earth Science Informatics*, *16*, 1993–2016.

STAC Contributors. (2021). https://stacspec.org (Abgerufen am: 02.06.2024).

The Kubernetes Authors. (2024a). Cluster Architektur. https://kubernetes.io/docs/concepts/architecture/ (Abgerufen am: 27.09.2024).

The Kubernetes Authors. (2024b). Overview. https://kubernetes.io/docs/concepts/overview/ (Abgerufen am: 27.09.2024).

Vögler, M., Schleicher, J. M., Inzinger, C., Dustdar, S., & Ranjan, R. (2016). Migrating Smart City Applications to the Cloud. *IEEE Cloud Computing*, *3*(2), 72–79.

Wilkinson, B., & Allen, M. (2005). *Parallel programming: techniques and applications using networked workstations and parallel computers* (2. Aufl.). Pearson Prentice Hall.

Wilkinson, M., Dumontier, M., & Aalbersberg, I. e. a. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*, 1–9.

Wu, Q. (2024). 13. Rioxarray. https://geog-312.gishub.org/book/geospatial/rioxarray.html (Abgerufen am 03.11.2024).

Xu, C., Du, X., Fan, X., Giuliani, G., Hu, Z., Wang, W., Liu, J., Wang, T., Yan, Z., Zhu, J., Jiang, T., & Guo, H. (2022). Cloud-based storage and computing for remote sensing big data: a technical review. *International Journal of Digital Earth*, *15*(1), 1417–1445.

Xu, C., Du, X., Jian, H., Dong, Y., Qin, W., Mu, H., Yan, Z., Zhu, J., & Fan, X. (2022). Analyzing large-scale Data Cubes with user-defined algorithms: A cloud-native approach. *International Journal of Applied Earth Observation and Geoinformation*, *109*, 1–11.

Yang, C., Yu, M., Hu, F., Jiang, Y., & Li, Y. (2017). Utilizing Cloud Computing to address big geospatial data challenges. *Computers, Environment and Urban Systems*, *61*, 120–128.

Yuan, M., & Mcintosh, J. (2002). A typology of spatiotemporal information queries. *Mining Spatio-Temporal Information Systems*, *699*, 63–81.

Zarr. (2024). Zarr. https://zarr.dev (Abgerufen am: 09.04.2024).

Zarr Developers. (2024). Zarr-Python Release 2.17.2.

Anhang

A Upload von COG-Dateien

```
1
2 def process_and_upload(rds, access, secret, count, chunk_size, com_typ="raw
    "):
3
4     endpoint_url = 'https://s3.fra1-1.cloudferro.com'
5     bucket_name = 'christina-rathjen-ma'
6
7     s3 = boto3.client(
8         's3',
9         aws_access_key_id=access,
10        aws_secret_access_key=secret,
11        endpoint_url=endpoint_url
12    )
13
14    # Definiere COG profile
15    cog_profile = cog_profiles.get(com_typ)
16    cog_profile.update(blockxsize=chunk_size)
17    cog_profile.update(blockysize=chunk_size)
18    cog_profile.update(
19        ALPHA="NO",
20        overview_level = 0,
21        forward_band_tags=True,
22        PREDICTOR=2
23    )
24
25    with MemoryFile() as memfile:
26        driver_options = {}
27        driver_options.update({
28            'blockxsize': chunk_size,
29            'blockysize': chunk_size,
30            'tiled': True
31        })
32
33        with memfile.open(
34                driver="GTiff",
35                dtype=rds.dtype,
36                count=4,
37                height=rds.rio.height,
38                width=rds.rio.width,
39                crs=rds.rio.crs,
40                transform=rds.rio.transform(),
41                **driver_options) as dataset:
42            if hasattr(rds, 'attrs'):
43                dataset.update_tags(**rds.attrs)
44
45            for i in range(1, 5):
46                band_data = rds.sel(band=i)  # Extrahiere das jeweilige
    Band
47                min_val, max_val, mean_val, stddev_val =
    calculate_band_statistics(band_data)
48                # Statistiken fuer das aktuelle Band
49                band_stats = {
50                    'STATISTICS_MINIMUM': min_val,
51                    'STATISTICS_MAXIMUM': max_val,
52                    'STATISTICS_MEAN': mean_val,
```

```
                    'STATISTICS_STDDEV': stddev_val
                }

                # Metadaten fuer das jeweilige Band aktualisieren
                dataset.update_tags(band=i, **band_stats)
                dataset.write(band_data.data, indexes=i)

        # Konvertierung zu COG
        with MemoryFile() as cog_memfile:
            cog_translate(
                memfile,
                cog_memfile.name,
                cog_profile,
                indexes=[1, 2, 3, 4],
                in_memory=True,
                quiet=True,
                forward_band_tags=True,
                overview_level=0,
            )

            # Upload in-memory COG
            s3_key = f"{count}_{rds.name}_{com_typ}_{chunk_size}.tif"
            s3.upload_fileobj(cog_memfile, bucket_name, s3_key)
            print(f"File uploaded to s3://{bucket_name}/{s3_key}")
```

Listing 16: Code zum Upload von COG-Dateien mit verschiedenen Kompressionsmethoden

B Raumzeitliche Abfragen mit COG-Dateien

```
def open_raster_from_s3(file_path, chunk, access, secret):

    def set_env_vars(vars_dict):
        for var_name,var_value in vars_dict.items():
            os.environ[var_name] = var_value

    ENDPOINT = "s3.fra1-1.cloudferro.com"
    vars_dict = dict(AWS_ACCESS_KEY_ID=access,
         AWS_SECRET_ACCESS_KEY=secret,
         AWS_S3_ENDPOINT=COS_ENDPOINT,
        AWS_VIRTUAL_HOSTING="False")
    set_env_vars(vars_dict)

    raster = rioxarray.open_rasterio(file_path).sel(x=slice(minx, maxx), y=
    slice(maxy, miny))

    time_str = raster.attrs.get('TIFFTAG_DATETIME', None)
    if time_str is not None:
        date_part, time_part = time_str.split(' ')
        date_part_corrected = date_part.replace(':', '-')
        timestamp = [pd.to_datetime(date_part_corrected).year]
        time_index = pd.DatetimeIndex([f"{year}-01-01" for year in
    timestamp], name="time")
        raster = raster.expand_dims({'time': time_index})
    return raster


ms = MemorySampler()
results = []
```

```
28
29 names = ['dop20rgbi_32_604_5748_2_ni_2016-08-24',
30  'dop20rgbi_32_604_5746_2_ni_2016-08-24',
31  'dop20rgbi_32_602_5748_2_ni_2016-08-24',
32  'dop20rgbi_32_602_5746_2_ni_2016-08-24']
33
34 paths = []
35 for name in names:
36     #path = f'christina-rathjen-ma/{count}_{name}_{com_typ}_10000.tif'
37     path = f's3://christina-rathjen-ma/{count}_{name}_partition_{chunk}.tif
    '
38     paths.append(path)
39
40 minx, maxx, miny, maxy = 602623.766578732, 5746418.865277786,
    605567.832769865, 5749401.924142947
41
42 client, gateway, cluster = createGatewayCluster(n_worker, memory, cores)
43
44 experiment_id = f"{count}_COG_{chunk}_{com_typ}_{time_points}_{ausschnitt}"
45 with ms.sample(f"Experiment {experiment_id}"):
46     with performance_report(filename=f"Abfragen_COG/Durchlauf1/bereich1/{
    experiment_id}.html"):
47         start_time = time.time()
48         rasters = [dask.delayed(open_raster_from_s3)(path, chunk) for path
    in paths]
49         datasets = dask.compute(*rasters, scheduler=client)
50         combi_dops = xr.combine_by_coords(datasets, combine_attrs="
    drop_conflicts", fill_value=np.uint8(0)).compute()
51         #combi_dops = xr.combine_by_coords(datasets, combine_attrs="
    drop_conflicts", fill_value=np.uint8(0)).persist()
52         #wait(combi_dops)
53         end_time = time.time()
54
55 duration, number_of_tasks, number_of_workers = extract_dask_data_from_run("
    Abfragen_COG/Durchlauf1/bereich1/",f"{experiment_id}.html")
56
57 # Speichern der Ergebnisse
58 experiment_result = {
59     'experiment_id': experiment_id,
60     'zugriffszeit_sek': duration,
61     'http_requests': s3.request_count,
62     'datenmenge': s3.total_bytes_sent,
63     'n_worker': number_of_workers,
64     'num_tasks': number_of_tasks,
65     'chunk_size': chunk,
66     'anzahl_dops': count,
67     'kompression': com_typ,
68 }
69 results.append(experiment_result)
```

Listing 17: Laden der COG-Dateien

C CPU-Auslastung beim Upload raumzeitlicher Rasterdateien

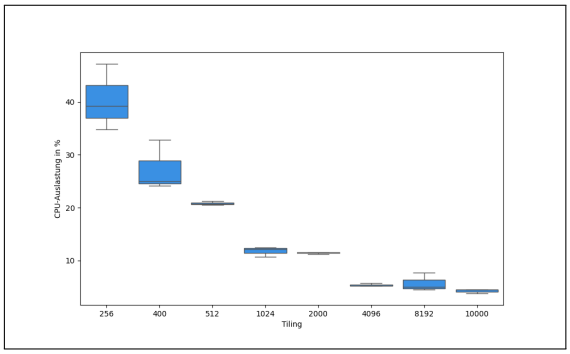

Abbildung 43: CPU-Auslastung beim Upload von COG-Dateien

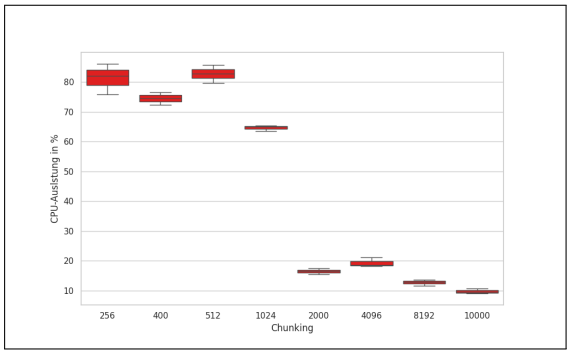

Abbildung 44: CPU-Auslastung beim Upload von Zarr-Dateien

D Berechnung raumzeitlicher Statistiken mit COG-Dateien

```
def calculateStatistics(ndvi_dataarray, threshold=0.25):
    # Extrahiere die Zeit aus den Attributen
    time = int(ndvi_dataarray.attrs.get('TIFFTAG_DATETIME', '').split(':')
    [0])

    # Berechnung der Statistiken pro Bild
    vegetation_mask = ndvi_dataarray > threshold
    vegetation_percentage = vegetation_mask.mean(dim=['y', 'x']) * 100
    mean_ndvi = ndvi_dataarray.mean()

    # Ergebnisse berechnen
    vegetation_percentage_value = vegetation_percentage.compute().item()
    mean_ndvi_value = mean_ndvi.compute().item()

    result = {
        'Jahr': time,
        'Mittlerer NDVI': mean_ndvi_value,
        'Vegetationsflaeche (%)': vegetation_percentage_value
    }
    return result

delayed_datasets = [dask.delayed(open_raster_from_s3)(path, access, secret)
    for path in matching_files]
delayed_ndvi = [dask.delayed(calculate_ndvi)(ds) for ds in delayed_datasets
    ]
delayed_result = [dask.delayed(calculateStatistics)(ds, threshold=0.25) for
    ds in delayed_ndvi]
results = dask.compute(*delayed_result)
df = pd.DataFrame(results)
# Berechnung des durchschnittlichen NDVI und der Vegetationsflaeche pro
    Jahr
yearly_summary = df.groupby('Jahr').agg({
    'Mittlerer NDVI': 'mean',
    'Vegetationsflaeche (%)': 'mean'
}).reset_index()
```

Listing 18: Berechnung des Jahresdurchschnitts und der prozentualen Vegetationsfläche mit COG-Dateien

E Berechnung raumzeitlicher Statistiken mit Zarr-Dateien

```
def calculateStatistics(ds, count, chunks, com_typ = None, threshold=0.25):
    time_points = ds['ndvi']['time'].values
    results = []
    for t in time_points:
        ndvi = ds['ndvi'].sel(time=t)
        vegetation_mask = ndvi > threshold
        vegetation_percentage = vegetation_mask.mean(dim=['y', 'x']) * 100
        time_str = np.datetime_as_string(t, unit='D')
        mean_ndvi = ndvi.mean()
        results.append({
            'count_images': count,
            'chunk':chunks,
            'com_typ': com_typ,
            'threshold' : 0.25,
            'Zeitpunkt': time_str,
            'Vegetationsflaeche (%)': vegetation_percentage.values,
            'Mittelwert': mean_ndvi.values
        })
    return results

results = calculateStatistics(ds_ndvi_persisted, count, chunk)
```

Listing 19: Berechnung des Jahresdurchschnitts und der prozentualen Vegetationsfläche mit Zarr-Dateien

F Raumzeitliche Abfragen mit Zarr-Dateien

	Chunking	**Kompression**	**Zarr / COG**					
Datenmenge (Anzahl DOPs)			**9**		**100**			
Testnummer			**1**	**2**	**3**	**4**	**5**	**6**
Räumlicher Ausschnitt			Klein	Klein	Klein	Klein	Groß	Groß
Anzahl der Zeitpunkte			1	3	1	3	1	3
512	512		5,24	10,94	8,31	15,99	24,65	50,26
1.024	1.024		3,17	6,97	6,93	10,82	19,92	32,68
2.000	2.000		2,52	5,91	6,93	10,82	19,92	32,68
4.096	4.096		2,72	6,59	8,64	10,12	21,77	31,10
10.000	Ohne		3,07	5,76	8,09	14,08	22,44	44,45
		Blosc ZSTD 9	3,44	6,53	7,48	10,81	19,75	30,38
		Blosc ZSTD 1	3,23	5,84	7,60	12,76	21,69	35,69
		Blosc LZ4 1	3,42	6,16	7,86	13,59	20,88	40,58
		Blosc LZ4 9	3,98	6,32	8,11	15,20	21,72	42,56
		LZ4 1	4,04	6,10	7,54	13,31	21,16	41,68
		ZSTD 1	3,39	5,96	7,34	12,41	20,15	31,90

Tabelle 13: Laufzeiten bei raumzeitlichen Abfragen mit Zarr für verschiedene Chunking- und Kompressionsmethoden

Dokumentation der Verwendung generativer KI-Systeme

Im Rahmen dieser Masterarbeit wurde das KI-System ChatGPT (Version 4.0) und Perplexity AI verwendet. Die Nutzung erfolgte in folgenden Bereichen:

- **Code-Entwicklung:** ChatGPT half bei der Erstellung von Python-Skripten, die zur Automatisierung von Arbeitsabläufen und zur Auswertung von Daten verwendet werden. Dies umfasste die Extraktion von Messergebnissen aus HTML-Dokumenten und das Überschreiben der S3fs-Klassen zur Zählung von HTTP-Requests. Die entsprechenden Codeabschnitte sind im Quellcode markiert. Dabei sei darauf hingewiesen, dass alle wesentlichen Entwicklungen und Hauptfunktionen eigenständig implementiert und durchgeführt wurden.
- **Fehlersuche und Debugging:** Bei der Identifikation und Behebung von Programmfehlern unterstützte ChatGPT und Perplexity AI durch Vorschläge zur Fehlerbehebung.
- **Textüberarbeitung:** ChatGPT diente dazu, den Text auf Rechtschreibfehler und grammatikalische Unstimmigkeiten zu überprüfen sowie Formulierungen zu optimieren und zu präzisieren, um die Verständlichkeit und den Lesefluss zu erhöhen.

Das Dokument „Erklärung gemäß dem für Ihren Studiengang gültigen Allgemeinen Teil (Teil A) der Prüfungsordnung an der Jade Hochschule Wilhelmshaven/ Oldenburg/ Elsfleth sowie zur Nutzung von KI-Systemen“ ist im digitalen Anhang hinterlegt.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die anderen veröffentlichten oder nicht veröffentlichten Schriften wörtlich oder sinngemäß entnommen wurden, habe ich kenntlich gemacht.
Die Arbeit hat in gleicher oder ähnlicher Fassung noch keiner anderen Prüfungsbehörde vorgelegen.

Datum, Ort Unterschrift