Technische Universität Berlin

Faculty of Electrical Engineering and Computer Science
Dept. of Computer Engineering and Microelectronics
Remote Sensing Image Analysis Group

An end-to-end framework for processing and analysis of big data in remote sensing

Master of Science in Computer Science

December, 2019

Viktor Bahr

Matriculation Number: 370681

First Supervisor:	Prof. Dr. Begüm Demir
Second Supervisor:	Prof. Dr. Tilmann Rabl
Advisor:	Gencer Sümbül

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Sämtliche benutzten Informationsquellen sowie das Gedankengut Dritter wurden im Text als solche kenntlich gemacht und im Literaturverzeichnis angeführt. Die Arbeit wurde bisher nicht veröffentlicht und keiner Prüfungsbehörde vorgelegt.

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 10.12.2019

. . . .

Viktor Bahr

Acknowledgements

I would like to thank my first supervisor, Prof. Begüm Demir, for her continued support and encouragement along the way. She has introduced me to remote sensing and thus opened up a highly interesting field of research for me, the further development of which I will follow with great excitement. I also thank my second supervisor, Prof. Tilmann Rabl, who supported me with his extensive knowledge of Big Data technologies on multiple occasions.

I thank the members of the RSiM group, and especially Gencer Sümbül, whose work I built on and who was always open to questions. Also, a thank you to Dr. Alireza Rezaei Mahdiraji of DFKI, who gave me important feedback.

My special thanks to Lutz Friedel of DIMA group for maintaining the computer cluster. You took your time for my requests even though you were very busy. I appreciate that. Also thank you Dr. Marcela Charfuelan for showing so much patience when sharing the cluster with me.

In addtition, thanks to all contributors to the numerous open source projects without which this work would not have been possible. Keep up the great work folks!

Finally my dearest thank you to my partner Annika, my friends and my family for consistently strengthening my back and dealing with my grumpiness. You mean the most to me, I couldn't have done it without you.

Abstract

Many types of remote sensing data have characteristics of *big data*. Particularly noteworthy are earth observation programs such as the Copernicus Program of the European Space Agency, which currently publishes approx. 13 TB of new data each day. At the same time, however, modern big data technologies are only slowly finding their way into the field, which is partly due to the lack of support for geodata. We present the implementation of a simple Big Data Framework for remote sensing data through the creation of a large-scale, multi-label image archive from multispectral satellite data. With Apache Spark, Hadoop, GDAL and PostGIS, we are using a number of popular, open source software packages. Our approach aims to cover the complete workflow, starting with data acquisition, through processing to analysis. We demonstrate those capabilities by acquiring 1.4 TB of satellite data from 39 European countries, processing them into 10.3 million image patches, and then determining their land cover labels. The insights gained from this work provide an impression of what needs to be considered when implementing a big data framework for remote sensing tasks and are thus suitable as a basis for further research.

Zusammenfassung

Viele Arten von Fernerkundungsdaten weisen Merkmale von *Big Data* auf. Hierbei besonders hervorzuheben sind Erbeobachtungsprogramme, wie z.B. das Copernicus Program der Europäischen Raumfahragentur, welches aktuell ca. 13 TB neue Daten pro Tag veröffentlicht. Gleichzeitung finden moderne Big Data Technologien aber nur langsam Einzug in das Feld, was unter anderem auf die mangelnde Unterstützung von Geodaten zuzückzuführen ist. Wir präsentieren anhand der Erzeugung eines umfangreichen, mehrfach annotierten Bilddatenarchivs aus multispektralen Satellitendaten die Umsetzung eines simplen Big Data Frameworks für Fernerkundungsdaten. Mit Apache Spark, Hadoop, GDAL und PostGIS greifen wir dabei auf eine Reihe von populären, quelloffenen Softwarepaketen zurück. Unser Ansatz zielt darauf ab den kompletten Arbeitsablauf abzubilden, angefangen bei der Datensammlung, über die Verarbeitung bis hin zur Auswertung. Diese Fähigkeiten demonstrieren wir, indem wir 1.4 TB Satellitendaten aus 39 europäischen Ländern sammeln, sie zu 10.3 Millionen Bildausschnitten weitererarbeiten und anschließend deren Bodenbedeckung und Landnutzunsdaten ermitteln. Die im Rahmen dieser Arbeit gewonnen Erkenntnisse liefern einen Eindruck darüber, was bei der Umsetzung eines Big Data Frameworks für Fernerkundungsdaten zu beachten ist und eignen sich so als Grundlage für weiterführende Forschung.

Contents

List of Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
DAG	Directed Acyclical Graph
CNES	Centre national d’études spatiales
CESBIO	Centre d’Etudes Spatiales de la Biosphère
DL	Download
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
EO	Earth Observation
GIS	Geo information systems
HDFS	Hadoop Distributed File System
JNI	Java Native Interface
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
L1C	Sentinel-2 processing level Level-1C
L2A	Sentinel-2 processing level Level-2A
MGRS	Military Grid Reference System
PEPS	Plateforme d’Exploitation des Produits Sentinel
PDGS	Payload Data Ground Segment
RDBMS	Relational Database Management System
RS	Remote Sensing
SQL	Structured Query Language
SRID	Spatial Reference Identifier
UDF	User Defined Function
UTM	Universal Transverse Mercator coordinate system
VCS	Version Control System

List of Figures

List of Tables

1. Introduction

1.1. Context

Today a number of earth observation programs provide open and free access to their data. Among them is the the European Space Agency's Copernicus program. Operational since 2014, it is made up of six dedicated satellite missions (The Sentinels) and around 30 contributing missions and aims to become the world's largest single earth observation program. The Sentinel missions include high -resolution radar and multi-spectral imaging for land, ocean and atmospheric monitoring. Individual missions are covered by satellite pairs for short revisiting times. While this comparatively high spatio-temporal resolution significantly increases the programs value for the scientific community it also aggravates a set of Big Data challenges: According to the Sentinel Data Access Annual Report 2018 [33] each day more than 13 TB of data is uploaded to the Copernicus portals. Even though the data is heavily standardized, the sheer size, high dimensionality and frequent updates pose additional challenges in storing, managing, processing, analyzing, visualizing and verifying the quality of data (Li et al., 2016 [43]). This is particularly true for remote sensing (RS) applications and has been the subject of multiple reviews (Li et al. 2016 [43], Ma et al. 2015 [45], Chi et al. 2016 [18]). Even though some efforts have been made to integrate geospatial data formats into existing cloud computing frameworks (e.g. Yu et al., 2015 [77] and Kini et al., 2014 [41]) much of the current RS literature is limited to locally computed, small-scale benchmark datasets.

The field of remote sensing (RS) is concerned with obtaining information about objects or areas from a distance, e.g. from satellites. As an example of the requirements of modern remote sensing research on a large-scale processing and analysis system, this work will concentrate on the work of Prof. Demir's Remote Sensing Image Analysis group at the TU Berlin. Established in 2018, the group's main focus has been the development of machine learning based image classification and retrieval systems for satellite data (e.g. Byju et al. [17], 2019; Sümbül et al., 2019a [65]). Out of need for large scale training datasets typically required by these specific machine learning techniques, Sümbül et al. published the *BigEarthNet* dataset [64] in 2019 - the first of its kind to be based on data from the multi-spectral Sentinel-2 satellites. Ground truth annotations are obtained by fusing the satellite data with land cover labels from the CORINE Land Cover (CLC) inventory (Bossard et al., 2000 [12]). Annotated satellite *tiles* are processed into smaller *image patches* and transferred to the a neural network framework for further analysis. As one limitation to this approach Sümbül et al. identified it's failure to scale with regards to the amount of available data. As a result of this, the current version of the BigEarthNet dataset contains only a subset of the area covered by the CLC inventory.

1.2. Problems & Strategies

Several aspects of remote sensing data processing and analysis pipelines can be considered as *big data* problems: Large volumes of data need to be acquired, processed and stored. Heterogeneous data sources need to be fused. Training and updating of machine-learning based analysis techniques require large amounts of verified training data and computation power. Hence, transferal to a big data processing context is expected to significantly improve performance as well as provide a fast and flexible foundation for further RS experiments. Over the course of this thesis we will explore contemporary distributed computing techniques and evaluate how they could help improve performance and overall usability for remote sensing processing and analysis workflows such as the one covered by BigEarthNet. Based on those findings we will implement a system prototype with the goal of migrating the BigEarthNet workflow to a distributed cloud computing context. This system will be divided in three parts: Data acquisition, processing and analysis. We will now continue to elaborate on the problems and opportunities at hand in more detail.

1.2.1. Data Acquisition

As of September 2019 the ESA's central archive for Copernicus data contains 22,336,732 published products. Since the start of operations in 2015, a total of 196.43 PB have been served to its 282359 registered users. To ensure open access and acceptable download speeds to its ever increasing user base, the archive's operators limit download speeds to 2.5 MB/s and two parallel downloads per user. Hence, long acquisition times can become an important bottleneck for users that plan to download large quantities of data. While a number of commercial redistributors offer considerably faster access to the Copernicus data this also imposes additional costs that might not be available for scientist from public research institutions.

Through concurrent access to a number of national mirrors (*Collaborative Ground Segments*) the data acquisition approach presented in Section 3.2 of this thesis aims to solve the problem of long acquisition times while maintaining the original acquisition workflow via OpenSearch / OData API endpoint and without imposing any additional costs. We introduce a number of fault tolerance techniques that attempt to solve problems specific to long-running, unattended acquisition sessions. By adding a distributed file system interface we try to integrate the data acquisition more strongly into a big data workflow.

1.2.2. Processing & Analysis

Although modern cluster computing frameworks typically provide a variety of parsers, algorithms, and data structures for the most common file formats and processing tasks, working with geospatial data typically requires the deployment of additional, non-native functions. Before those functionalities can be used in a cloud computing context the following requirements need to be met:

- Compatibility with the cloud computing framework of choice
- Compatibility with worker node hardware and operating system requirements
- A common channel for distributing and setting up 3rd-party methods on each of the worker nodes

To help broaden the scope off possible applications the processing cluster in question should be built with commonly-used, highly available and mature software packages to allow for an easy setup and high reproducibility. Furthermore, the effort required to migrate non-distributed processing workflows to a distributed computing framework should be minimized so that researchers need to spent as little time as possible on migrating their previously established codebase. One problem often encountered in remote sensing research is storage and integration of data with different geographic coordinate systems which will also need to be addressed by our proposed system.

In Section 3.3 we propose a system for the distributed generation of large-scale image archives from Sentinel-2 data. We will identify a distributed processing engine that meets the above requirements and extend it with the capabilities for working with geospatial data. By introducing a high-level concept for development and deployment of distributed processing jobs we aim to make the framework more accessible. A distributed storage will be used to store the satellite data as well as the generated image archive. For the generation of the image archive, different tasks need to be performed. We will divide them into a distinct processing and analysis jobs. Metadata produced in these jobs is stored in a common database which allows the jobs to be related to each other. Since we are mainly working with geospatial data, we choose a database that provides adequate support for the storage of such data. In Section 3.4 we propose a technique for the large-scale fusion of data from external sources, such as the CORINE Land Cover inventory, with the generated image archive. We first review a list of geospatial query engines and then present a design for integrating such capabilities with the existing framework.

1.3. Contribution

The main contribution of this work is the implementation of a big data framework for remote sensing tasks using a number of popular open-source projects. For each of the three sections of our framework, we review a list of existing approaches and point out ways to utilize them for remote sensing data. We work out important design decisions and explain how we approached them. In addition, we identify weak points in the presented approach and discuss ways to solve them. These findings have the potential to serve as a basis for future research. In contrast to the existing literature, we do not limit ourselves to a single section, but rather examine the complete end-to-end workflow.

1.4. Structure of the Thesis

The thesis will be structured as follows: We begin by introducing a number of technologies that will be used in the course of this thesis in chapter 2. In chapter 3 we propose a system design for our end-to-end framework prototype. In this context, we divide the framework into three subsystems: acquisition, processing and analysis. Following this structure we elaborate on the details of the system implementation in chapter 4. In the chapter 5 we evaluate the performance of the individual systems and the framework as a whole on the basis of data sets of different sizes. Finally, in Chapter 6, we discuss the results of the evaluation and summarize the insights gained during this work.

2. Fundamentals

In this chapter we will introduce a number of technologies and services that will be used in the course of this thesis.

2.1. Sentinel-2 Data

The Sentinel-2 mission is part of the Copernicus Earth Observation program. The twin satellites Sentinel-2A and B generate multi-spectral data with 13 bands in the visible, near infrared, and short wave infrared spectrum. Depending on the band type a spatial resolution of 10m, 20m and 60m is provided. Revisiting time is around 5 days. Images generated by the satellites are stored together with auxiliary data as *products* in a standardized folder structure called *SAFE*.

2.1.1. Atmospheric Correction

The process of atmospheric correction describes the conversion of top-of-atmosphere to bottom-of-atmosphere reflectance values in satellite data. For Sentinel-2 data, the degree of processing is indicated with the help of distinct *processing levels*. The following algorithms provide support for atmospheric correction of Sentinel-2 data:

- iCor [24]
- MAJA [31]
- ARCSI [14]
- Sen2Cor [44]

2.1.2. Sentinel-2 Processing Levels

In their Sentinel-2 handbook [3], the European Space Agency defines three data *processing level*: Level-0, Level-1 and Level-2. The raw sensor data is received by one of the *Payload Data Ground Segment*s' (PDGS) X-band core stations and forwarded to the *Processing and Archiving Centers* for Level-0 and Level-1 processing. Since December 2018 Level-2 processing are also systematically generated by the PDGS.

Level-0

Level-0 products consist of compressed image data and related metadata (ephemeris, attitude data, thermal data etc.). They are not released to users but form the basis of further processing steps.

Level-1

Level-1 processing is divided into three parts: A, B and C. Level-1A processing focuses on de-compression source data. Level-1B processing includes radiometric corrections like removal of defective pixels, de-convolution and de-noising. Imaged data is organized in multiple 25 km by 23 km *granules* in JPEG2000 format (Christopoulos et al., 2000 [19]). Level-1C processing includes ortho-rectification and spatial registration on a global reference system. Top-Of-Atmosphere (TOA) radiance values are calculated and cloud, land and water masks generated. Image data is organized in 100 km by 100 km tiles in UTM WGS84 projection. Level-1C is the lowest processing level available through the Open Access Hub. The average size of a Level-1C product is 600 MB [3].

Level-2A

Level-2A processing includes a Scene Classification as well as the calculation of bottom-of-athmosphere radiance values from Level-1C data. Additional outputs include an Aerosol Optical Thickness map a Water Vapour map and a Scene Classification map together with Quality Indicators for cloud and snow probabilities. Level-2A processing can also be performed by the users with the `Sen2Cor` processor (Louis et al., 2016 [44]) offered by the ESA. The average size of an Level-2A product is 800 MB [3].

2.2. Copernicus Open Access Hub

The *Copernicus Open Access Hub* provides free and open access to the data produced by the Sentinel missions. Besides a graphical-user interface, which allows self-registered users to do full-text searches, select custom areas-of-interest and perform product previews it also offers script-based product search and download via `OpenSearch` and `OData` API endpoints.

2.2.1. Application Program Interfaces

Two separate APIs are used to provide access to the Copernicus data:

- *OpenSearch (Solr)* for product discovery
- *OData* for data access

OpenSearch (Solr)

The OpenSearch interface allows searching the Copernicus product catalog. In addition to simple attribute filters it also supports geographical search queries (i.e. via Well-known text representation [39]). It follows the *Representational state transfer* (REST) architectural paradigm: Search queries are passed as a `GET`-request to the `/search` endpoint. Product information is returned in standardized JSON payloads. The Copernicus product discovery backend is powered by the *Apache Solr* [27] search engine, a popular open source enterprise search platform that supports distributed indexing.

OData

The OData [66] interface provides access to the Copernicus product data. While it can also be used as a less complex complement to the OpenSearch product discovery endpoint it is mainly used for previewing and downloading satellite data. As the discovery interface it also follows the *Representational state transfer* (REST) paradigm: A paginated list of products is returned by the `/Products` endpoint. Detailed product information can be accessed via the `/Products('Id')` endpoint. Sensor data can be obtained from `/Products('Id')/$value`.

2.2.2. Collaborative Ground Segments

Collaborative ground segments provide complementary access to Copernicus data and/or to specific data products or distribution channels. Segments are run by state-actors and international organizations independent of the Copernicus program. Areas of cooperation range from participation in data acquisition and production over support for validation/calibration activities and application development to redistribution of Sentinel products via local mirror sites [3]. As the official Copernicus mirror (Copernicus Open Access Hub) limits bandwidth due to high traffic volumes the national mirrors are especially interesting for speeding up access to the Sentinel data. At the time of writing the following ground segments were actively mirroring data:

Country	**standard API specs**	**URL**
Australia	No	`https://copernicus.nci.org.au/sara.client/#/home`
Austria	Yes	`https://data.sentinel.zamg.ac.at/#/home`
Finland	Yes	`https://finhub.nsdc.fmi.fi/#/home`
France	No	`https://peps.cnes.fr/rocket/#/home`
Germany	No	`https://code-de.org/`
Greece	Yes	`https://sentinels.space.noa.gr/`
Italy	No	`http://collaborative.mt.asi.it/#/home`
Norway	Yes	`https://colhub.met.no/#/home`
Portugal	No	`https://ipsentinel.ipma.pt/dhus/#/home`
UK	No	`http://sedas.satapps.org/#tools`

Table 2.1.
Listing of Collaborative Ground Segments

As shown in Table 2.1, the majority of the ground segments (6/10) deviate from the standard Copernicus API specifications, which excludes them from use with one of the Copernicus API clients.

2.2.3. Clients

In addition to a graphical user-interface the Copernicus Open Access Hub, as well as many Collaborative Ground Segments, provide data access via the OpenSearch and OData protocols. Also, a number of private companies maintain mirrors of the Copernicus data on Amazon Web Services. While direct access via a download manager such as GNU wget [73] is possible, a number of open-source projects aim to provide a more convenient interface by wrapping the employed protocols used for searching and downloading the data. Table 2.2 lists all actively maintained projects. Due to external requirements I will limit my subsequent analysis to mirrors not using AWS.

Project	OpenSearch	OData	AWS	Language
sentinelsat [61]	Yes	Yes	No	Python
sentinelhub [60]	Yes	Yes	Yes	Python
sat-api [59]	No	No	Yes	JavaScript
peps_download [51]	No	No	No	Python
awsdownload [42]	No	No	Yes	Java

Table 2.2.
Open-source libraries for acquisition of Copernicus data

2.3. CORINE Land Cover Inventory

First launched in 1985 by the European Commission, the CORINE (Coordination of Information on the Environment) program aims to compile and standardize information on the state of the environment for its member countries. This includes information on the type of environment, land cover structure and geographical distribution. It is generated in a bottom-up fashion - First, data is collected on a national level and then integrated into a joint database.

The minimum mapping unit is 25 ha, with a minimum width of 100m and a positional accuracy of better than 100m. It aims to provide a thematic accuracy of $> 85\%$ at an scale of 1:250000 [22].

CLC 2018

The fifth edition of the CORINE Land Cover inventory includes 44 land cover classes and data from 39 member states. A list of land cover classes and covered countries can be found in the Appendix C.1. It was collected over the one-year period of 2017-2018 and is available in both raster and vector format from

`https://land.copernicus.eu/pan-european/corine-land-cover/clc2018`

2.4. Eurostat NUTS

The Nomenclature of Territorial Units for Statistics (NUTS) is a standard developed by the European Statistical Office Eurostat for describing the subdivision of countries. It defines three distinct levels: NUTS 1 describes "*major socio-economic regions*", NUTS 2 "*basic regions for the application of regional policies*" and NUTS 3 "*small regions for specific diagnoses*" [26]. In its latest 2016 version, the dataset includes the 28 EU member countries, 5 EU candidate countries and 4 EFTA countries. It is available as vector data from

`https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/`
`administrative-units-statistical-units/nuts`

2.5. Geospatial Data

In geospatial data, data is extended by location information. An illustrative example is provided by the satellite data used in this work, in which measured reflection values are assigned to the area of the recording by means of *coordinates*. A *coordinate reference system* is used to describe how to relate a three dimensional location information to a two-dimensional map coordinate.

Raster Data

Spatial information represented in a grid of pixels is called *raster data*. Each pixel has its own value. Raster data can contain both discrete and continuous information. The accuracy of the data is dependent on the grid size. Satellite images are an example of raster data.

Vector Data

Spatial information represented as nodes and edges is called *vector data*. There are three main types of vector data: points, lines and polygons. A line is created by connecting points, a polygon by connecting lines to form an enclosed area.

Geospatial Data Abstraction Library

The Geospatial Data Abstraction Library (GDAL) [72] is a software library for handling a variety of vector and raster data formats. First published in 2000 by F. Warmerdam it has become the de-facto standard for working with geospatial data on all major platforms. Written in C++, it provides an API for the Python and Java programming languages and includes a range of command line tools for data translation and processing.

2.6. Spatial Data Management

Over the course of this thesis we will generate large volumes of geospatial data, mostly associated with satellite images. In order to manage this information independently of the pixel data, we use a spatial data management system based on a relational database management system.

PostGIS

PostGIS [56] is an extension to the PostgreSQL (Momjian, 2001 [49]) object-relational database that adds support for geospatial data types and operators. In addition to storing metadata we will use its high-performance query planner for fusing geospatial data from multiple sources.

2.7. Distributed Computing

In distributed computing, a problem is divided into several smaller problems that are solved simultaneously on different computers. Those computers, often referred to as *nodes*, are networked with one another in *clusters* and share information by exchanging messages. A number of software frameworks offer distributed calculation functions. These frameworks can typically be divided into systems for distributed storage, processing and analysis.

2.7.1. Apache Hadoop

The Apache Hadoop Framework [4] is an open-source framework for distributed storage and processing of large volumes of data based on the map-reduce paradigm (Dean et al., 2008) written in Java. It includes the Hadoop Distributed File System (HDFS) for distributed storage of data, Hadoop YARN (Yet Another Resource Negotiator) for managing cluster resources and Hadoop MapReduce for defining large-scale distributed processing jobs.

Hadoop follows a master-slave architecture. In HDFS, the master node is called the *Name node*, the slaves are called *Data nodes*. When a file is uploaded to the HDFS, it is divided into multiple *blocks*. Fault tolerance is introduced by replicating the blocks on a number of Data nodes. The Name node keeps track of all blocks stored in the HDFS. For MapReduce, the master is called the *Job tracker* and slaves are called *Task tracker*. MapReduce jobs are divided into tasks which are delegated to Task trackers along with a reference to the respective blocks.

Access to files stored in the HDFS can be gained through the native Java API, HTTP and the command line interface. Clients for a variety of other programming languages can be generated with Thrift (Slee et al., 2007 [63]).

2.7.2. Apache Spark

Apache Spark (Zaharia, 2016 [78]) is an open-source framework for large-scale distributed processing and analysis. Spark was designed to address shortcomings of Hadoop MapReduce and adds support for processing successive jobs *in-memory*. A number of libraries implemented on top of the Spark processing engine add support for SQL, machine learning and stream processing. Spark does not include a cluster resource manager and a distributed file system. However, a wide range of popular platforms such as the Hadoop ecosystem are supported. Spark is written Scala. Bindings for the Java, Python and R programming languages are available.

Spark also follows a master-slave architecture. The master daemon launches a *driver process* that executes the parallel operations defined by the user. Through the cluster manager, the driver process first acquires a number of slaves, so called *executors*, and then divides the user application into smaller *tasks*. These tasks are then forwarded to the executors.

All Spark operations are based on the Resilient Distributed Dataset (RDD) data structure, a read-only multiset of distributed items. RDDs are organized into *partitions*, which are stored on a distributed file system. Two types of operations are supported by RDDs, *transformations* and *actions*. A transformation functions translates one RDD into another. An example of such a transformation function is the `map` operation, where a user-defined function is applied to a all partitions of an RDD. Transformations are only executed once an action function is called. An example of an action function is the writing of transformed RDDs into the distributed file system. Based on the user-defined RDD operations, the driver process generates an execution plan that is modeled as a directed acyclic graph with partitions as nodes and transformations as edges.

The SparkSQL library introduces a number of additional, high-level data types as well an optimized SQL query engine. A *DataFrame* is the conceptual equivalent of a table in a relational database and can either be generated from existing RDDs or external data source. The SparkSQL query engine supports execution of SQL queries against a data frame. Internally, DataFrames consist of RDDs, so the RDD API can still be used. While user-defined functions (UDF) applied through the RDD API are *black boxes* for Spark's optimization engine, the SQL algebra is supported by a variety of powerful optimization strategies.

2.8. Hardware Setup

Our cluster consists of 9 nodes: 8 nodes each with 25 AMD Opteron 6238 (2.6 GHz) CPUs and 257947 MB RAM and one node with 50 Type 6238 CPUs and 499867 MB RAM. The latter node acts as our Spark and Hadoop master. On every node 16 hard drives with 1.8 TB capacity each are mounted in a HDFS data node. The maximal capacity of the HDFS is 214.8 TB.

3. System Design

3.1. Overview

In this chapter we present our design for an end-to-end framework for processing and analysis of big data in earth observation. Following the challenges described in Section 1, the framework is divided into three parts:

1. A system for the swift **acquisition** of large volumes of open access satellite data
2. A system for conveniently **processing** large volumes of remote sensing data in a cloud computing cluster
3. An extension to the processing system that enables large-scale **analysis** of satellite data

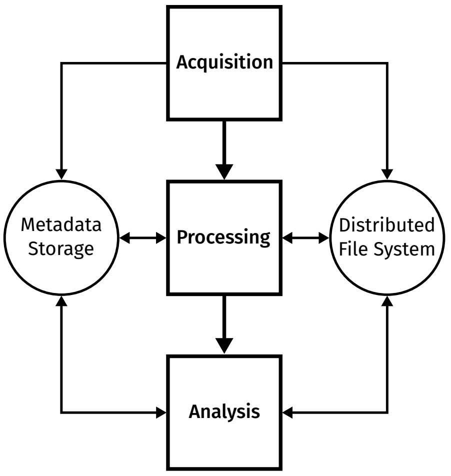


Figure 3.1.

The three stages are connected through a common data management scheme. Sensor data is stored on a distributed file system. Metadata is either handled in a flat file database or uploaded to an SQL database.

We reviewed a number of contemporary big data technologies and adapted them for use in our design. Special attention was paid to the use of open source software to keep the framework as accessible as possible. Also, usability and user-centricity were emphasized throughout the design process. With this in mind we developed a *workflow template* to simplify the use of cluster processing technologies for remote sensing research.

We demonstrated the capabilities of the framework by generating a large-scale, multi-label *benchmark archive* from Sentinel-2 data: Large volumes of Sentinel-2 data were acquired. Sentinel-2 data was split into smaller *image patches*. Image patches were annotated with land cover, country and scene classification labels

3.2. Data Acquisition

3.2.1. Requirements

As listed in Section 2.2.3 a number of software packages provide access to the Sentinel data. However, downloading large amounts of data from all over the European continent, as is required for our specific use-case, creates a unique set of problems addressed by neither of the available packages:

Copernicus data is available from a number of sources (see Section 2.2.2). As all open-access distributors of Copernicus data limit the available bandwidth and number of connections per user, users are faced with a fixed upper-bound on download speed which is specific to the respective service and independent of the locally available connection quality.

By far the largest[1] amount of Copernicus data is available from at the *Open Access Hub (SciHub)*, centrally run by the European Space Agency. While half of the national mirrors outperform the *SciHub* mirror, the scope of available data is at the same time typically limited to the borders of their respective operators' nation state. Also, data retention times are often significantly reduced for those mirrors.

Mirror	Avg. DL (MB/s)	Conn.
Austria	1.2 ± 0.1	2
ESA (SciHub)	2.8 ± 0.8	2
Finland	2.5 ± 1.1	4
France	10.1 ± 0.6	4
Germany	50 ± 1.7	2
Greece	1.5 ± 0.1	4
Norway	48.5 ± 3.2	1

Table 3.1.
Copernicus mirror average download speed ($n = 10$) and maximal number of allowed connections. Measured 2019-01-15 at TU-Berlin. Mirrors with technical problems were excluded.

Users with a demand to maximize both download speed and area covered by the products available are thus faced with a dilemma: Either sacrifice speed over tempo-spatial coverage or vice versa. The approach to be designed should therefore be able to meet both of the aforementioned requirements. Furthermore, it should address a number of additional requirements that arise from our specific use case:

- Fault-tolerance techniques to prevent failure of long, unsupervised downloads
- Custom product filters (e.g. lowest cloud cover per UTM grid and season)
- Statistics for product selection (e.g. minimal / maximal cloud coverage)
- Storage of product metadata for later use
- Provide convenient access to a distributed file system for further processing

3.2.2. CollGS: Concurrent Access to Multiple Mirrors

In order to satisfy the above requirements we propose abstracting an existing, well-established client to support parallelized mirror access, thus providing simultaneous access to both high-speed and high-coverage mirrors while reusing the proven workflow of a non-parallelized client. We named this proof-of-concept implementation `CollGS` after the network of Copernicus **Coll**aborative **G**round **S**egments (see Section 2.2.2) it uses to increase the number of potential data sources.

[1]In terms of tempo-spatial coverage.

To create it we proceeded as follows:

1. From the list of existing clients, we selected the one most suitable for extension to parallel mirror access
2. While keeping as much of the original client's workflow intact as possible, we implemented a fault-tolerant mechanism for accessing multiple mirrors in parallel
3. We developed a scheduling strategy for concurrent downloads to ensures optimal utilization of the available bandwidth
4. We addressed further use-case specific requirements

3.2.3. Selecting an Existing Client for Abstraction

From the list of existing Copernicus API clients (see Section 2.2.3), we chose the most suitable client for implementing parallel mirror access based on the following criteria:

- Easily abstractable, provide well encapsulated access to API endpoints
- Actively maintained (number of commits in the last 12 months)
- Popularity, as an indicator for high usability (number of GitHub stars)
- (Optional) High test coverage (coverage percentage)

Rank	**Project**	**API access**	**Commits**	**Stars**	**Test coverage**
1.	sentinelsat	Yes	75	365	93%
2.	sentinelhub-py	Yes	132	320	88%
3.	sat-api	No	239	127	No
4.	peps_download	No	4	51	No
5.	awsdownload	No	0	17	No

Table 3.2.
Parallelization feasability ranking of available Copernicus API clients.

Both the `sentinelsat` and `sentinelhub-py` packages provide well-encapsulated access to standard Collaborative Ground Segment APIs. While `sentinelsat` comes with a monolithic `SentinelAPI` class as a joint interface to the OpenSearch and OData endpoints, `sentinelhub-py` is organized in a more modular structure. Dedicated API interfaces are placed in separate classes alongside a number of auxiliaries. Both the `sat-api` and `awsdownload` packages were organized similarly, but are limited to data hosted on AWS. The `peps_download` script does not support the standardized Copernicus API schemata, but only a custom variety provided by the French Sentinel mirror, PEPS.

Judging by the number of commits between June 2018 and April 2019, `sentinelsat`, `sentinelhub-py` and `sat-api` seem to be actively maintained projects. The same three packages also collected more than 100 stars each. The currently most popular client on GitHub is `sentinelsat` with 365 stars, followed by `sentinelhub-py` with 320. Both libraries included comprehensive unit tests.

The evaluation of existing clients yielded two candidates that met all our requirements: `sentinelsat` and sentinelhub-py. While sentinelhub-py included a variety of useful features, its modular design also made it more complicated to modify, with many core features scattered across the package index. We thus chose the more concise sentinelsat package for implementing concurrent mirror access.

3.2.4. Concurrent Mirror Access

The sentinelsat package encapsulates all logic used for interacting with OpenSearch / OpenData endpoints in a class called SentinelAPI. For the CollGS parallellized acquisition tool we propose a high-level wrapper class (SentinelAPIManager) which holds an array of SentinelAPI instances, one for each mirror, and re-implements the methods offered by the SentinelAPI class in a parallelized fashion. By re-using existing methods for interacting with the API endpoints from a well maintained and tested third party library, the effort required to achieve parallel product search and acquisition is reduced to introducing and managing concurrency. Although the nature of the changes required for this varied across methods (i.e. download method requires custom scheduling logic) they can be generally summarized under the following points:

1. Mirror login data, input parameters are parsed
2. For each mirror *one thread* is spawned, a base class (SentinelAPI) instance is created and a callable containing the respective API call is submitted for asynchronous execution to a thread pool
3. On completion, threads are joined, results collected and further processed (i.e. search results are merged, products decompressed)

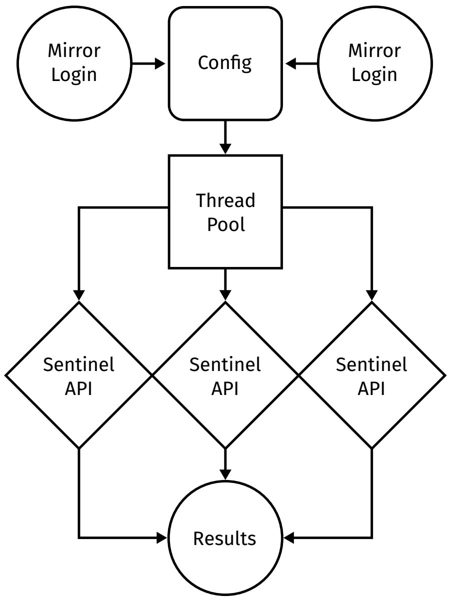


Figure 3.2.

In line with the strategy outlined above we parallelized all SentinelAPI class methods offering access to API endpoints. To protect against typical concurrency issues, such as race conditions, thread-safe data structures were used. Potential errors, like timeouts and response validation failure, were addressed by adding a retry mechanism to the threads: Callables that raise an exception are retried until a user-defined threshold is reached. To avoid failure of subsequent requests the respective mirror is temporarily excluded from the list of available data sources once this limit is passed. When a failed call is retried successfully the retry counter is reset.

By introducing fault tolerance to the concurrent API call execution, we made the workflow resilient against outage of individual mirrors. Furthermore, failing downloads no longer lead to the termination of the entire session which is particularly useful when acquiring large quantities of data in lengthy and unsupervised sessions.

3.2.5. Download Scheduling

Concurrent mirror access introduces a number of challenges specific to the scheduling and management of product downloads:

- Collaborative ground segment providers impose limits on both the number and the rate of connections; Multiple mirrors should be used in parallel in order to increase aggregate download speed
- Users might also want to impose a limit on the total number of active downloads to accommodate for their local connection properties
- Differences in mirror performance need to be considered in the scheduling process; Better performing mirrors should be preferred
- The duplication of product downloads must be avoided; The state of a download should be unambiguous
- Scheduling strategy should be able to adapt dynamically to failing downloads and mirror outages

We selected a finite state machine as the most suitable paradigm for implementing the unambiguous download state management. Figure 3.3 depicts the state transition flow. Download state machine instances, one for each requested product, are registered with a *schedule manager daemon*. The schedule manager daemon continuously checks the registered instances for state changes, manages open connection, updates the download schedule and assigns active download jobs to the thread pool. Once running inside an individual thread the download job autonomously executes the sequence of transitions defined by the state transition flow. Following the retry strategy outlined in Section 3.2.4 failed download are rescheduled by the schedule manager daemon until a user-defined threshold is reached. For each mirror the number of retries is monitored and once a user defined limit is exceeded the mirror is excluded from the scheduling process. Products that are exclusively available from the affected server are reverted to the *failed* state.

Before commencing with a download session CollGS users are asked to compute a performance ranking for the mirrors they have set up. Based on a list of user-defined MGRS grid zone identifiers the following parameters are measured: Download speed, request latency, number of products available for the given test zones and number of all products available at the respective mirror. According to the weighting defined in Table 3.3 the measurements are then used to calculate a performance score against which the mirrors are ranked. The calculated score as well as the measured parameters are stored on disk for later use.

Metric	**Weight**
DL Speed	1.5×
Test Products	1×
Total Products	1×
Response Time	0.5×

Table 3.3.
Mirror performance score weights

The download schedule defines the order of execution as well as the data source to use. On registration with the schedule manager daemon each download job is assigned a list of data sources that offer the respective product. In case performance rankings are available for the given data sources, scheduled download jobs are ordered by their cumulative mirror performance score. This ensures

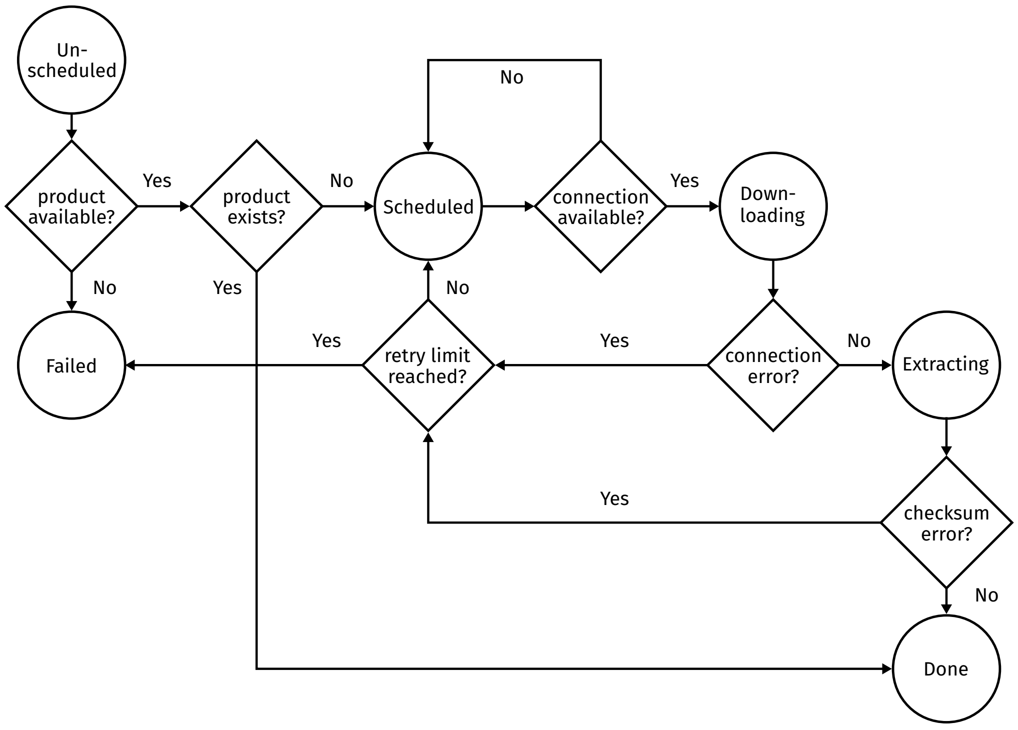


Figure 3.3.
Download state transition flowchart. States are shown as circles, decisions as diamonds.

that highly available products are executed first, thus enlarging the potential bandwidth footprint [2]. Already existing products are immediately moved to the *done* state. Products that are not available on any configured mirror are forced to transition to the *failed* state. After each cycle of state updates the schedule manager daemon iterates over the list of products and checks their respective sources' number of open connections. Once a server with free slots is encountered and the cumulative number of open connections falls below the user-defined limit, the associated download job is submitted to the thread pool for execution. If, at that time, more than one data source is available the one with the greater performance score is selected. Products for which no free slot could be found are skipped until the next update cycle.

When a download is completed successfully the download speed is reported and the zipped product is forwarded to a dedicated *process* for decompression. Data integrity is validated by comparing the checksum of the downloaded file with the one provided by the original API response. Corrupted files are reported, removed from disk and re-scheduled.

3.2.6. Custom Filtering & Product Statistics

Many collaborative ground segments offer an API endpoint based on the OpenSearch standard for product discovery (see Section 2.2.1). Request made to and responses received from those endpoints

[2]High availability increases chances of downloading from multiple sources and hence increasing the number of potential connections. Furthermore, these additional connections would not be subject to rate limits imposed by servers with already running downloads.

are mutually compatible. At the time of writing, the returned JSON payloads included 32 metadata fields which, among others, allowed identifying a product's acquisition date and location, satellite mission and sensor, as well as file properties. To help users narrow down the scope of available products, discovery endpoints by default support simple filtering queries on these fields. More complex query operations, like aggregation and sorting, were not supported by the OpenSearch standard and thus had to be implemented on the client side.

Researchers from the RSIM group for example were interested in selecting tiles with the lowest cloud cover percentage per UTM zone and season acquired over a specific date range. The following custom filtering logic was implemented in the CollGS prototype: After filtering all products for the given time frame using the OpenSearch endpoints the returned JSON responses are collected and merged by their unique identifier, as multiple mirrors might offer the same product. The list of tile metadata is then grouped by UTM zone and season. The resulting groups are each sorted by cloud cover percentage[3] and finally the product with the lowest cloud cover percentage is selected.

The JSON responses returned by the discovery endpoints can also be used to calculate client-side statistics on the selected products. In reference to the above-mentioned custom filter the CollGS prototype implements the following metrics: Number of selected products, average product size, cumulative product size, number of UTM zones selected, average number of products per zone, average cloud cover percentage and number of products per season.

By combining custom filtering and product statistics the presented approach offers a convenient way to evaluate and adapt the scope of a download session with regards to a number of custom requirements and *before* starting the actual data acquisition. Given the sheer size of the Copernicus archives such techniques can have a profound impact on the time spent with identifying data suitable for further analysis.

3.2.7. Metadata Storage

Since no special requirements for the data format resulted from the CollGS design specifications, the exchange of metadata between different subsections of the data acquisition stage (e.g. between product discovery and product download) was based upon the standardized JSON response object returned by the OpenSearch endpoints. The same light-weight format was also used to connect the results of the acquisition stage to all subsequent processing stages. As an alternative approach, metadata storage in a relational database was evaluated but eventually rejected since the requirements placed on the storage back-end could also be solved without integrating such complex dependencies. To accommodate users that still prefer to use a relational database for metadata storage an auxiliary script for inserting the JSON data into a SQL table was provided. With respect to the additional information gained during the course of the acquisition process the original schema (see A.1.1) was extended by two fields: Season of the year and product location on disk. The latter field is optional and only becomes populated after successful acquisition of the respective product. In order to avoid redundant and potentially time-consuming queries tile metadata is cached in between acquisition subtasks. After successfully completing a download session the merged JSON objects are used to create an index of the obtained products.

3.2.8. Distributed File System Interface

Modern-day big data workflows typically rely on a *distributed file system* (DFS) to provide fast, fault-tolerant and transparent access to large amounts of data. One example for such a system is the popular

[3]Provided by OpenSearch endpoint response.

Hadoop Distributed File System (HDFS) (see Section 2.7.1). As the CollGS prototype specifically targets users that plan to acquire large quantities of satellite data, a distributed file system interface for the convenient upload of obtained products was implemented. Due to its high prevalence and explicit support for low-cost commodity hardware the interface was based on the Apache Hadoop stack. Information regarding the files to be uploaded are either gained by indexing a local directory or by specifying a JSON file following the structure defined in Section 3.2.7. Similar to the approach described in Section 3.2.6, a number of filters allow the user to limit the scope of the upload to the distributed file system. After successfully uploading a batch of products, the respective metadata is updated with the location of files on the DFS.

3.3. Processing

3.3.1. Requirements

Remote sensing workflows typically require a number of (pre-)processing steps to be applied to the acquired imagery before further analysis can be conducted. In the case of the large-scale dataset generation task described in Section 3.1, this includes the following tasks:

1. Conversion of acquired images to Level-2A products
2. Generation of non-overlapping *image patches* from Level-2A products
3. Export of image patch dataset to file

The nature of the above tasks allows us to design a system based on parallelization. While the atmospheric correction can be applied to every product individually, image patches can be generated from separate sensor bands[4]. File exports can be performed on a per-patch basis. A parallelized system is particularly useful for addressing problems caused by large volumes of input data: Distributing the workload reduces the overall read / write overhead and makes it more easy to address a job's specific resource requirements. No specialized hardware is needed as even the largest jobs can be run on appropriately dimensioned clusters of commodity hardware. With the rise of the *cloud computing* paradigm scaleable on-demand solutions for obtaining storage and processing resources have become highly available and widely used in both commercial and research applications.

Although new earth observation data is published with high velocity, images used in the generation of annotated datasets are typically collected over long time periods (i.e. years). This lack of need for rapid responsiveness led to the assessment that *batch processing* rather than *stream processing* was the most appropriate processing mode for the given research question.

A number of cloud computing software libraries provide data structures and algorithms suitable for distributed batch processing. As we wanted to avoid re-implementing a major part of the existing codebase for manipulating Copernicus data, we narrowed our search to frameworks with support for the Python programming language. At the time of writing, the following actively maintained open source projects were available:

[4]Patches from separate bands need to be merged in a later step.

- `PySpark`
  A Python interface for Apache Spark [78] (see Section 2.7.2)

- `Dask`
  A flexible parallel computing library for analytics [23]

- `mrjob`
  A Python MapReduce library [37]

After an initial feature comparison `mrjobs` was removed from the list due to missing in-memory computation capabilities. Because of the overhead caused by communication between the Python interpreter and the Java Virtual Machine, we initially expected PySpark to perform worse than Dask. A search of the existing comparative literature did not confirm this suspicion. In their 2019 performance comparison Dugré et al. [25] found that "*despite slight differences between Spark and Dask, both engines perform comparably*" for data-intensive neuroimaging pipelines. Since version 2.3 Spark includes Apache Arrow [5], a "*cross-language development platform for in-memory data*", which offers a common memory format for both Python and Java applications and thus helps to further reduce the overhead created by PySpark. Initial small-scale tests confirmed that the two APIs were designed similarly and that both were suitable options for implementing the task at hand. After consolidating with the administrators of the local server cluster we decided to build on their previous experience and use Apache Spark in combination with the Hadoop Distributed File System for implementing the processing stage.

Capabilities for working with geospatial data in general, and for manipulating Copernicus data in particular, are neither included in Apache Spark nor in the Python standard library. We thus had to first identify a suitable third party software, and then organize its deployment the worker nodes. We identified two types of alternatives: Python libraries and Apache Spark add-ons. Almost all of the Python interfaces were based on the GDAL (Warmerdam, 2008 [72]) library, which is the cornerstone for interacting with geospatial data in many programming languages. The same can be said about the Apache Spark add-ons, which typically implement geospatial Spark data structures on top of the GDAL Java bindings. We compared the two groups in order to select the alternative most suitable for implementing our processing tasks:

While the available Spark add-ons provide comprehensive, often optimized, support for geospatial data types, many of the features are aimed at high-level analysis. Also, it is unclear to what extend our approach would benefit from these additions, as many of the processing task will be defined as *User Defined Functions* which are not subject to optimization by the Spark query processor. Python interfaces are rarely supported, hence many of the existing processing tools we plan to use are at risk of having to be re-written in a JVM compatible language. Required third party dependencies, however, are already included in the add-ons. Also, deployment of add-ons to cluster is typically well supported by Spark.

Using a low-level Python library on the other hand, would require that we implement our own geospatial data structures and Spark parallelization logic. While this requires additional planning, it also gives us full control over the *complexity* of the implemented approach. In contrast to the Spark add-ons we could tailor data structures exactly to the requirements of the processing job. However, we would also need to devise a strategy for deploying the required libraries to the worker nodes.

Similar to the choice of the processing framework, initial tests confirmed that both routes of adding geospatial capabilities to Spark could be used to implement the processing stage. In an effort to *keep* *it simple stupid*[5] we chose to implement our own geospatial data structure prototypes using a low-level Python library. After evaluating the available packages we quickly chose the GDAL Python bindings which we were already familiar with.

To avoid a time-consuming manual dependency setup, we designed a strategy based on the Hadoop YARN *resource manager* for automatic resolution and deployment of dependencies to the worker nodes on job submission. This strategy was embedded in a concept to simplify usage and development of PySpark jobs. The proposed *workflow template* was designed based on the following requirements:

- Management and deployment of dependencies should be designed as convenient as possible, built on existing techniques
- The ability to specify dependencies globally (i.e. for all jobs) and locally (i.e. for single jobs)
- Transparent exposure of job parameters through configuration files
- Modularized code structure, clean encapsulation of business and parallelization logic to allow for an easy migration of existing code
- Allow easy debugging and unit testing

3.3.2. PySpark Workflow

As no official recommendations regarding the structure of a PySpark workflow are specified by the developer team, a number of concepts have been proposed by the user community. Inspired by E. Kampf's 2017 essay *Best Practices Writing Production-Grade PySpark Jobs* [40] we defined a *big data workflow* to be comprised of:

- **Business logic**
  Describes how to process or analyze the data; SparkSQL or User Defined Function; May include calls to 3rd party dependencies
- **Parallelization logic**
  Describes how to parallelize the business logic and to what elements it should be applied to; Spark transformations are typically defined here
- **Initialization logic**
  Describes how to initialize the cluster context, load configuration files, start the parallelization logic and finally, store results
- **Configuration**
  Variable parameters used to configure a job
- **Tests**
  Unit tests to validate if the workflow works as excepted
- **Utilities**
  Tools for managing the workflow

[5]Refers to the KISS design principle coined by Lockheed lead engineer Kelly Johnson in 1960 [32]

Following this definition we propose a modular structure for organizing our PySpark workflow: Dedicated tasks are arranged in folders called *job modules*. Job modules are placed inside the workflow's `jobs/` directory. The initialization logic forms the base of a module and is placed in a standardized `job.py` file at the module's root. YAML-formatted configuration parameters are placed in a `config.yml` file. Business and parallelization logic are put into Python modules and also placed inside the job folder. Whenever possible, the runtime-optimized SparkSQL algebra should be preferred over User Defined Function to define the business logic. The parallelization logic interface exposed to the initialization script should be designed as concise and abstract as possible to allow for clean encapsulation of logic levels. Third party dependencies are specified in a `pip` compatible `requirements.txt` file. Unit tests are added to the `tests/` directory.

Organizing PySpark jobs in standardized modules allows us to design management utilities that can be applied to every such job. Through the common `config.yml` channel, job configuration can be done transparently and without cluttering up the submission call to the cluster manager. By strictly separating the logic levels we improve readability and enable the convenient transfer of existing logic fragments to the PySpark ecosystem. Migration of pre-existing business logic, for example, would not require any changes to the business logic itself but only the creation of the appropriate initialization and parallelization logic. Existing dependency specifications can be re-used, as we are implementing standard Python dependency management techniques.

3.3.3. Dependency Management

PySpark job dependency management can be divided into two steps:

1. Local dependency specification
2. Deployment of specified dependencies to the worker nodes

As mentioned above, in standardized PySpark job modules specification of third party dependencies can be done using the *requirement file syntax* introduced by `pip`, Python's de-facto standard package manager. In addition, all Python modules placed inside a job module are assumed to contain business or parallelization logic and will also be submitted to the worker nodes. To avoid redundancy, dependencies that are used by more than one job module can be declared *global* either by specifying them in a `requirements.txt` file placed at the root of the workflow directory or by adding them to the `dependecies/` folder, also located at the workflow's base directory. As for some non-native libraries `pip` only provides the respective Python interface but not the required sytem-level binaries, we added additional support for the `conda` [36] package manager. This open source package manager, developed by Anaconda Inc., differs from `pip` in that it offers packages for multiple programming languages and typically includes system-level binaries. `conda`'s environment files are handled in the same way as `pip`'s requirement files - They can either be placed inside job modules or in the workflow's base directory.

At the time a job is started, its dependencies need to be present on all connected nodes. The PySpark community has developed a number of strategies to address this, all of which are centered around bootstrapping *isolated Python environments*. Spark itself offers a method for adding files from a distributed file system to the executor's *Spark context*. Among others, F. Wilhelm demonstrated the distribution of PySpark dependencies based on this technique in his 2018 blog post on *Managing isolated Environments with PySpark* [74]. A slightly different approach is taken by Hadoop's YARN ressource manager: It spawns Spark executors inside virtualized containers and allows users to add files to the container context before launching the job. Spark's YARN interface allows submitting compressed Python virtual environments, so called *archives*, alongside of job scripts. The archives are uploaded to the HDFS and extracted to all containers created by YARN.

In combination with the package managers introduced above, both presented techniques allow us to dynamically configure the context a job is executed in. As we were already planning to use YARN as a cluster manager, we chose the latter option for our workflow. To simplify the generation of job environments we created the `build.sh` utility. It accepts job module names as input argument and installs the respective jobs' business logic, parallization logic and the specified dependencies into a *conda virtual environment*[6]. The generated environment is then compressed to a tar file and moved to a dedicated folder (`envs/`) in the workflow's base directory. This abstraction of both dependency discovery and resolution in one build command allows a fast and easy preparation of deployments.

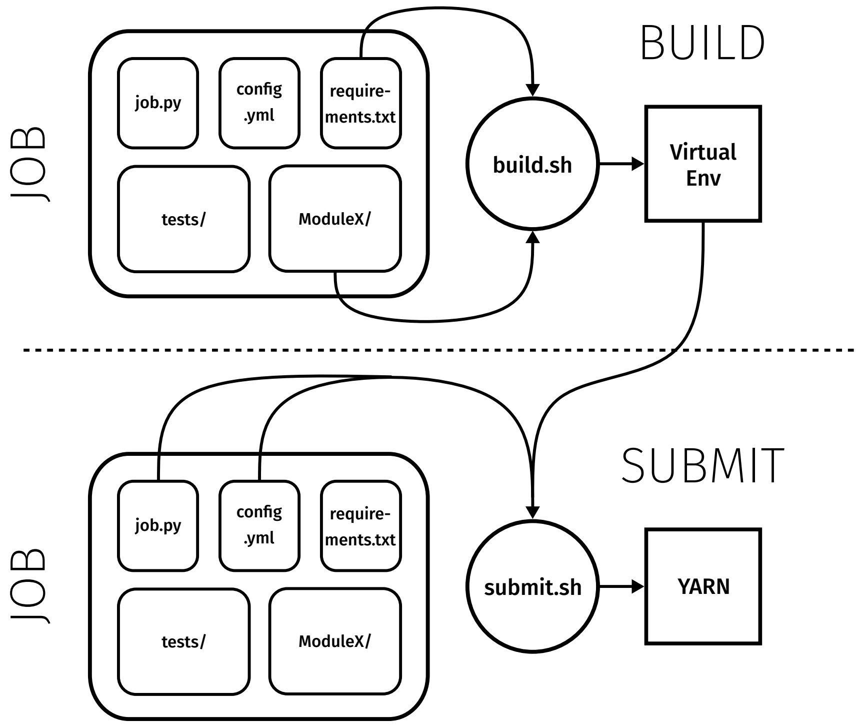


Figure 3.4.
Exemplified PySpark job build and submit workflow

3.3.4. Deployment

PySpark applications are typically deployed via the `spark-submit` command line interface provided by Apache Spark. It offers a variety of options for configuring Spark itself, the associated cluster manager and the respective job. As jobs grow more complex so do the calls to the submission script. To avoid having to specify long and complicated `spark-submit` calls on every deployment, we introduced the `submit.sh` utility to our workflow. Analog to the `build.sh` script it accepts

[6]The choice for `conda` environments was made, because they are compatible with `pip` but not vice versa.

job module names as input argument and submits the jobs' initialization script, configuration file and virtual environment to the cluster. Additional `spark-submit` parameters, like the number of executors to spawn or which environmental variables to set, are also placed into the utility. Custom logic for abstracting the most common groups of deployment options are defined using *shell script*.

After adapting the job layout described in Section 3.3.2, executing PySpark jobs in a cluster is reduced to three simple steps:

1. Obtain a copy of the job code (e.g. by pulling from a VCS)
2. Call `build.sh` to generate the virtual environment
3. Call `submit.sh` to submit the job to the cluster

For changes that only affect the initialization logic or the configuration file, Step 2. can be omitted. A graphical representation of the *build & submit* workflow is shown in Figure 3.4. A detailed list of options specified to the `spark-submit` tool can be found in Section 4.3.2.

3.3.5. Distributed Atmospheric Correction

Interacting with the earth's atmosphere changes the property of light that is reflected from the earth surface to the optical sensors of the Sentinel mission. Before further analysis can be conducted, the imagery needs to be *atmospherically corrected*, that is, the true surface reflectance values need to be computed. As described in Section 2.1.1, a number of processors offer atmospheric correction of Sentinel-2 products. Besides correction of reflectance values, these processors typically include a number of additional processing steps, like terrain and cirrus correction and cloud detection. However, none of the presented options offer native support for distributed processing.

Selecting an Atmospheric Correction Algorithm

Following the workflow concept introduced in Section 3.3.2 we chose to implement distributed atmospheric correction as a PySpark job. From the list of existing algorithms we chose the one most suitable for being included in such a job based on the following criteria:

- Portability
- Compatibility with the PySpark ecosystem
- Configuration overhead
- Quality of documentation

iCOR

Developed by the Belgian remote sensing company VITO, the iCOR [24] processor offers atmospheric correction of both Sentinel-2 and Landsat-8 imagery. A free version of the processor is distributed as a plugin for ESA's *Sentinel Application Platform* (SNAP), a Java-based collection of *toolboxes* for working with earth observation data. Configuration is exclusively handled via the SNAP user interface. No developer documentation is provided with the free version.

Due to the closed-source nature of the plugin and it's dependency on SNAP, migrating iCOR to a PySpark job was deemed unfeasible.

MAJA

The *MACCS ATCOR Joint Algorithm* [31], in short MAJA, is an atmospheric processor for LANDSAT and Sentinel-2 products developed by the French space agency CNES in collaboration with the German Aerospace Center, DLR. The main difference to its competitors is the use of *time series* data to improve cloud detection performance. Written in C++, it is available for download in the form of a pre-compiled, Linux-only binary. While some semi-official tutorials document usage of MAJA, no official documentation is provided. Similar to ARCSI, the default Sentinel-2 SAFE file format is replaced in favor for the custom MUSCATE format, which is developed by CNES. While processing parameters were not included in the original package, we were able to obtain working examples from a third party website.

Albeit MAJA's usage of multi-temporal data helps improve the cloud detection accuracy, it also requires processing in time-aligned batches, which would reduce the level of parallelism that files could be processed at. In combination with the closed-source nature of the distributed package, the lack of proper documentation and the use of a custom output format we decided against using MAJA in our distributed atmospheric correction.

ARCSI

The open source Python package for *Atmospheric and Radiometric Correction of Satellite Imagery*, ARCSI [14], is developed by P. Bunting at Aberystwyth University, Wales. It supports a number of sensor types (Sentinel-2, Landsat, Rapideye, SPOT and more) and includes adequate documentation. Processing parameters for Sentinel-2 products need to be provided by the user. Notably, it is highly reliant on a range of rather unknown third party formats and libraries. By default, products are converted to the KEA file format (P. Bunting et al., 2013 [15]), a modification of the popular HDF5 format. The bulk of the processing is done with the help of the remote sensing and GIS software library *RSGISLib* (P. Bunting et al., 2014 [16]).

While we assessed ARCSI to be compatible with PySpark, it also seemed to require adapting a number of specialized technologies, like the KEA file format, that are not compatible with more commonly used varieties. As this could potentially restrict all further processing steps to the capabilities offered by the aforementioned technologies, we chose to eliminate ARCSI from the list of candidate processors.

Sen2Cor

The Sen2cor package [44] is the official processor used by the European Space Agency for generating Sentinel-2 Level-2A products (see Section 2.1.2). Including a wide range of features, like scene classification and cloud detection, it is distributed both as a SNAP plugin and a standalone installer. The standalone installer generates an isolated Python environment. A comprehensive developer manual is provided, as well as default configuration files for Sentinel-2 images. While most of the codebase is open source, the key atmospheric correction stage, developed by the German Aerospace Center, is patent protected and hence only available as pre-compiled binary.

Job Design

Although somewhat limited by the mode of distribution and the patent protection, the Sen2Cor processor was deemed to be a suitable candidate for implementing our distributed atmospheric correction job. No additional configuration parameters had to be specified - Correction operations could be started right after installing. Other than its closest competitor, ARCSI, it does not introduce a special output format, but reuses the SAFE file format Level1-C images are stored in.

Also, processing is based on a number of well-known and well-maintained Python libraries. We devised a number of tools, described in Section 4.3.4, to isolate the respective Python module from the standalone installer and deal with the patent protected binaries. Furthermore, we added support for the Hadoop Distributed File System. This enabled us to use the Sen2Cor processor for the distributed atmospheric correction job outline below.

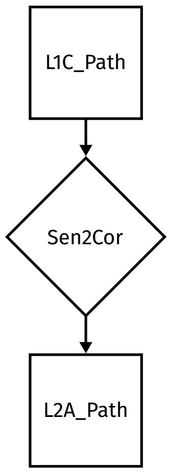


Figure 3.5.

Having selected the Sen2Cor processor, we moved on to designing the PySpark job: Information about the products to be converted are read from a `CollGS` JSON files which is specified in the job's configuration file. After obtaining a list of input product paths and their respective UUIDs, the initialization logic starts the Spark driver process, partitions the data and schedules distribution to the executors. Then, the parallelization logic is invoked. The modified Sen2Cor processor is applied to all partitions of the input data. To ensure fault-tolerance, failed conversions are retried for a user-specified number of time and then ignored. The path of the final L2A product, returned by the modified processor, is associated with it's original L1C product and forwarded to the parallelization logic. The parallelization logic collects the output from all partitions and returns them to the initialization logic. There, completed partitions are written to a user-specified parquet file. As an additional way of storing the output, the initialization logic allows registering metadata of completed products with a SQL database.

3.3.6. Distributed Patch Generation

One application for big data in remote sensing is the generation of training data for machine learning models. While the benefits of advanced machine learning techniques, like deep learning, for remote sensing tasks have been demonstrated in numerous publications over the recent years, relatively few publicly available datasets are able to satisfy their high demand for training data adequately. With *BigEarthNet* [64], Sümbül et al. presented the first large-scale, multi-label dataset derived from Sentinel-2 data. Following this example, we will demonstrate how to generate an even larger Sentinel-2 based archive, using a range of big data technologies.

In the the previous section we have demonstrated how to convert a large set of raw satellite images into analysis-ready data. Before we can continue with obtaining ground-truth annotations, the issue of product size must be addressed: Analysis-ready Sentinel-2 data (Level-2A) is stored in $100 \times 100km^2$ tiles - Way to much information for any model to handle at once. Hence, the Level-2A products must first be split into smaller *image patches*.

In this section we will propose a strategy for generating large volumes of image patches from Level-2A data, using PySpark. As with the above pre-processing stage, this strategy will be based on the big data workflow concept introduced in Section 3.3.2.

Job Design

We started the design process for the patch generation job with a definition of the desired output data. The process should create a number of *image patches* including the following attributes:

- References to the input data (L1C & L2A product), including a 2-dimensional *patch index* to indicate a patch's position in the original L2A tile

- A collection of sensor *bands*, including:
  - References to the band name and path
  - The sensor band data
  - Information about the band's pixel resolution
- Coordinates to identify the location covered by the patch
- Spatial Reference System Identifier (SRID)
- MRGS grid identifier

Both L1C and L2A products are stored in the SAFE format - A directory structure for organizing sensor data, or *image granules*, and additional metadata. Due to the multi-spectral nature of the Sentinel-2 sensor, a granule is comprised of multiple[7] *sensor bands*, which are stored in the JPEG2000 format (Christopoulos et al., 2000 [19]). Before we are able to load the actual sensor data, we first have to identify the relevant band files from a given SAFE path. A list of band types to be ignored can be provided via the configuration file. While we could, in theory, also discard references to the input SAFE after identifying the sensor bands, we chose to include it in the patch data for subsequent quality control and because it allows introducing band-level parallelism to the generation process:

Parsing large volumes of data is prone to cause large memory footprints. Instead of having to load all of a tile's bands at once, which could exceed the available computing capacity and lead to idle times, we can apply patch generation on each sensor band individually and thus distribute the load more evenly. The resulting patches can than be grouped by L2A path and *patch index*. The patch index is returned bv the generator and indicates the window offset used to extract the patch from the original image. For very large jobs, however, aggregation operations like this can lead to an excessive amount of *shuffles*, i.e. redistribution of data across partitions, which would significantly impact performance. Under such circumstances, product-level parallelism could become viable again. Hence, we will offer switching between both strategies in the job's configuration file.

Following the structure of the SAFE file, an image patch contains multiple sensor bands. References to the original sensor band file are stored to ensures that every patch can be traced back to it's original input. The band name helps to identify the type of data. Besides an atmospherically corrected versions of the bands, originally included in the L1C file (B01 - B12), our patches will contain a copy of the scene classification band (SCL) generated by Sen2Cor. All other bands will be ignored in order to limit the size of the dataset. As multi-dimensional array types are not well supported by Apache Spark, the band data has to be stored in a one-dimensional vector.

Sensor bands are recorded at three different resolutions: 10m, 20m or 60m per pixel. As the original *BigEarthNet* dataset, we will use fixed-size, non-overlapping sliding window for generating the patches. The pixel resolution determines the patch size: For 10m bands $120 \times 120 pixel$ are extracted, for 20m $60 \times 60 pixel$ and for 60m bands $20 \times 20 pixel$. Capabilities for reading the JPEG2000 format are added to PySpark by including the GDAL (Warmerdam, 2008 [72]) library in the job's requirements file.

Together with a Spatial Reference System Identifier, patch coordinates give an unambiguous description of the patch's location. This will in the following sections be used to integrate additional

[7]While Level-1C products include 13 original sensor bands, Sen2Cor adds additional Scene Classification, Aerosol Optical Thickness, Water Vapour and Quality Indicator bands to the Level-2A products. Also, existing bands are resampled at different resolutions, resulting in 40 bands per L2A tile.

data with the patches. While the SRID can be copied from the original tile, the patch coordinates need to be re-calculated. The MGRS grid identifier, extracted from the sensor band metadata, assigns the patch to a $100 \times 100 km^2$ grid zone and is mainly used for grouping spatially related patches.

Having defined our desired output data, we can continue designing the PySpark job. As Spark's data structures are immutable, we will need to define a series of *transformations* in order to create the output structure. We propose grouping them in a common `Sentinel2PatchGenerator` class, with parallelization logic as class methods that calls the business logic. This allows accessing the transformations individually, e.g. for debugging, as well as exposing a single interface, which connects all transitions, to the initialization logic. The job is kept concise and every level of logic is focused on it's intended purpose.

The following transformations were defined:

1. From the input data, Level-2A product paths, relevant sensor bands are identified. Output: Band information with associated L2A path.
2. From each sensor band, patches are generated. Sensor bands are read into memory and split, patch coordinates are calculated and remaining patch metadata is copied. Output: Patches with data from one (band-level parallelism) or more (product-level parallelism) bands, including associated L2A path and patch coordinates.
3. Sensor band patches are aggregated by patch index and L2A path. This is not required when bands were loaded at once (product-level parallelism). Output: Final patch objects with data from multiple bands.

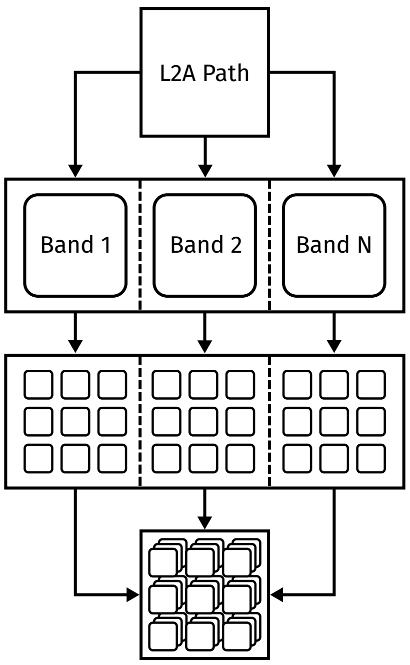


Figure 3.6.

The initialization logic parses the configuration parameters and schedules loading of the input data. On starting the above transformations, partitions with L2A product paths are distributed to the worker nodes. Completed partitions are passed from one transformation to another. Before the final aggregation stage can be started, all partitions are required to complete the patch generation stage. After the final transformation, the generated patches can either be passed on to the subsequent analysis stages (see Section 3.4) or written to a user-defined parquet file. Patch metadata can be uploaded to an SQL database. A comprehensive summary of the patch generation job implementation, along with an example SQL schema, can be found in Section 4.3.5.

3.3.7. Distributed Data Export

To demonstrate the capabilities of our end-to-end framework, we chose to generate a large-scale image dataset from Sentinel-2 data. Up until now, this dataset is stored in a columnar PySpark data structure, called `DataFrame`. However, by distributing the data in this Spark native we risk excluding many potential users. The Spark ecosystem provides a variety of connectors to export `DataFrames` into more commonly used formats, like Parquet, JSON, or SQL tables. But without additional modifications, none of the available options were able to meet our requirements for publishable remote sensing datasets:

- Patches are available as image files
- The image format of choice is widely supported and allows storage of geospatial metata, such as map projection and coordinate system
- Additional metadata is stored in a common data-interchange file format, close to the image

In order to meet those requirements, we designed a PySpark job for distributed export of the image patch dataset: First, we selected an image format that is both widely supported and allows us to store georeferencing information alongside the sensor data. While many common image formats, like JPEG (Wallace, 1992 [71]), support adding geo-coordinates in the form of *geo tags*, they typically limit functionality to GPS coordinates with *World Geodetic System* (WGS84) reference coordinate system. More comprehensive support for geospatial metadata is only available from specialized formats.

GeoTIFF (Ritter & Ruth, 1997 [58]) is one such format. It allows embedding a range of geospatial information inside a TIFF file and is widely used among the remote sensing community. TIFF (ISO 12639:2004 [29]), short for *Tagged Image File Format*, is a popular format for storing raster graphics. Libraries to access TIFF files are available in many major programming languages. Although access to the geospatial data tags still requires a specialized parser, GeoTIFF pixel values can also be read with standard TIFF parser. In summary, GeoTIFF represents the best compromise between accessibility and specialization and was thus chosen as the output format for our image patches. Capabilities for writing GeoTIFF files are added to PySpark by including the GDAL (Warmerdam, 2008 [72]) library in the job's requirements file.

As the data-interchange format for storing non-geospatial metadata we chose the *JavaScript Object Notation* (Bray, 2014 [13]), JSON for short. While more readable and lightweight alternatives exist, e.g. YAML (Ben-Kiki et al., 2005 [10]) or TOML [67], they typically require deploying an additional library for reading the data programmatically. For JSON, however, native support is offered by most major programming languages, including Python.

Before we could continue designing the export logic, one last technical problem had to be addressed: HDFS is not designed to handle large amounts of small files efficiently. As every object is kept in memory by the Hadoop *namenode*, a high number of objects, smaller than the HDFS block size of 64MB, can cause excessive memory consumption and thus degrade the namenode's performance. To avoid this issue, known as the "*Small Files Problem*" [8], we do not export patch files to the HDFS directly, but aggregate them into archive files before uploading.

With all technical questions resolved, we can move on to designing the export logic: We will start with a PySpark `DataFrame` containing images patches, similar to the structure defined in Section 3.3.6. Depending on the level of processing, additional columns might be present, for example land cover labels added in Section 3.4.3. We will need to account for this when exporting the patch metadata. On submission to the cluster manager, the initialization logic parses the job configuration and schedules loading of the input data. Then, the parallelization logic is invoked and applies a UDF to all partitions of the `DataFrame`. The band data is reshaped from a Spark vector back to a matrix, and combined with the geospatial metadata. Following the structure of the original BigEarthNet dataset (Sumbul et al., 2019 [64]), *single band* GeoTIFF files are written to *patch directories*. Patch directory names are generated by combining the first three elements[8] of the L2A SAFE name with the patch index. GeoTIFF file names are built by adding the name of the respective band to the patch directory name. Available patch metadata is written to a JSON file and placed along the images. Patch directories are organized by MGRS grid zones. Once all elements of a partition have been exported, they are added to a zip file archive. The name of the archive is generated from the partition index. Completed zip files are moved to the HDFS, while local copies of the patch directories are removed.

HDFS zip file paths are collected by the parallelization and returned to the initalization logic. In case a local directory for unpacking the archives is specified, the Spark driver process spawns a number of threads and starts concurrently downloading and decompressing the archives into the specified directory. Once extracted, the acquired zip files are removed by the driver node.

More information regarding the implementation of the distributed data export job can be found in Section 4.3.6.

3.4. Analysis

3.4.1. Requirements

In the previous sections we have designed a system for acquiring and processing large volumes of satellite data into a dataset of image patches. In order to increase the value of the dataset, we will now proceed with analyzing the patch data.

Similar to the processing, the nature of the analysis tasks allows us to design a parallelized system. Analysis can be conducted on every image patch individually, independent of the other patches. We will thus base our design on the *PySpark workflow* introduced in Section 3.3.2. To perform the tasks described below, we will use both capabilities offered natively by Apache Spark, and those provided by third party libraries.
Generally speaking, additional insights about the data can be gained from two types of sources:

1. **Internal**
   Information that is already contained in the patch data can be further analyzed

2. **External**
   Information from external datasets can be integrated with the patch data

Accordingly, we designed two distinct example tasks: To demonstrate the utilization of internal data, we extracted the distinct label values from the scene classification band (SCL), converted them into readable strings and added them to patch attributes. Also, they were used to filter out patches that contained unwanted scene labels. For demonstrating integration of external data, we obtained patch land cover labels and country information, using the associated georeferencing information. As support for such geospatial queries is not included in Apache Spark, we first had to select a viable alternative processor and then incorporate it into to the PySpark logic.

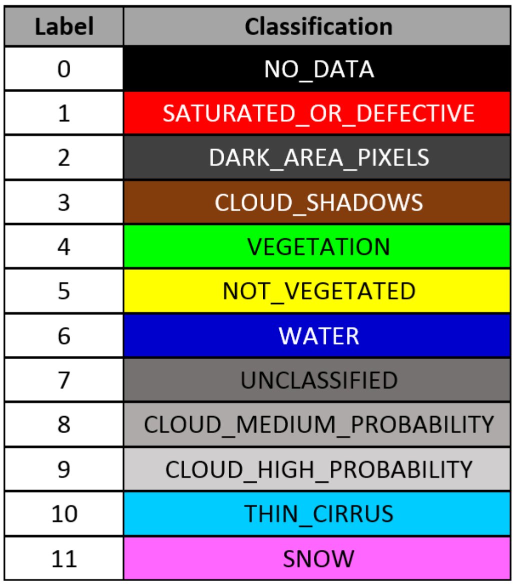
[8]Sentinel mission type, processing level and original acquisition date

As processing and analysis are typically performed together, the above tasks were integrated into the `Sentinel2PatchGenerator` class, proposed in Section 3.3.6. To enable further analysis of patch metadata independently of the pixel data, support for uploading it to an SQL table is provided.

3.4.2. Scene Classification Label Annotation

On a closer look, both processing and analysis is performed in the distributed atmospheric correction job described in Section 3.3.5:

When converting Level-1C into Level-2A products, the Sen2Cor processor not only performs an atmospheric correction, but also a scene classification of the satellite data. Combined thresholds are applied to the sensor bands and a classification mask, containing one of 12 distinct scene labels for each pixel, is generated. The mask is then exported as a JPEG2000 file and stored together with the other bands. Pixel resolution of the classification map is set to 60m. The available scene labels are depicted in Figure 3.7.



Label	Classification
0	NO_DATA
1	SATURATED_OR_DEFECTIVE
2	DARK_AREA_PIXELS
3	CLOUD_SHADOWS
4	VEGETATION
5	NOT_VEGETATED
6	WATER
7	UNCLASSIFIED
8	CLOUD_MEDIUM_PROBABILITY
9	CLOUD_HIGH_PROBABILITY
10	THIN_CIRRUS
11	SNOW

Figure 3.7.
Level-2A Scene Classification Values [2]

In the patch generation, the scene classification mask is imported along with the other bands and split into patches. Hence, scene classification data is available for every patch at the end of this stage. In order to obtain scene class labels, a distinct set of patch scene class values is computed and converted to label string using the mapping defined in Figure 3.7. The resulting labels are added to the respective patch. If desired by the user, all patches containing the `NO_DATA` label[9] are filtered out. The resulting `DataFrame` is either passed on to the next analysis stage or returned to the initialization logic.

3.4.3. Land Cover & Country Label Annotation

The geospatial metadata associated with the patches provides an excellent foundation for the integration of external data sources. In this section, we will propose a design for fusing data from two external sources with the patch data, using the patch geolocation. Since geospatial queries are not supported by Apache Spark natively we first identified a number of open-source systems that provide this feature. The resulting list can be split into two categories: Systems that extends Apache Spark and standalone processors.

Geospatial Apache Spark Extensions

The following open-source packages add support for processing and querying geospatial data to Apache Spark v2.4:

[9] An indicator for missing sensor data.

- **GeoSpark**
  The GeoSpark package (Yu et al., 2015 [77]) extends Spark's default distributed data structure `RDD` with georeferencing information to form Spatial `RDD`s (`SRDD`). Based on `SRDD`s, a number of query optimized geospatial operations are implemented. Supports both vector and raster data. Java, Scala, R and Python interfaces are available.

- **GeoTrellis**
  Written is Scala, the GeoTrellis (Kini & Emanuele, 2014 [41]) package provides various data structures and algorithms for processing large geospatial raster datasets in Apache Spark. Database connectors add support for storing raster data on various distributed database systems. Python bindings are available under the name *GeoPySpark*.

Standalone Geospatial Query Engines

The following actively maintained query engines provide support for geospatial data:

- **GeoMesa**
  Written in Scala, the GeoMesa software suite (Hughes et al. 2015 [35]) aims to enable large-scale geospatial querying and analytics on a number of popular distributed database systems, like Accumulo, HBase and Cassandra. Spark bindings for analyzing data stored in GeoMesa are also available. Support for raster data is deprecated.

- **MongoDB**
  The document-oriented *NoSQL* database MongoDB (Banker, 2011 [9]) database includes native support for geospatial queries on data stored in the GeoJSON format.

- **PostGIS**
  An extension to the PostgreSQL (Momjian, 2001 [49]) object-oriented RDBMS, PostGIS (Ramsey, 2005 [56]) adds extensive support for spatial objects and query operations to the SQL language.

Selecting a Geospatial Processor

The listed alternatives were then assessed for their suitability with regard to their use in our analysis. Particular emphasis was placed on the performance of spatial join operations, as this was expected to be the most heavily used type of operation for integrating additional data with our patches. However, several factors made the evaluation difficult for us:

There is little comparative literature on the subject. To our knowledge, none of the relevant publications covers all of the systems mentioned above. Also it was not possible for us to carry out tests in advance due to the limited timeframe of this thesis. Therefore, our results can only be regarded as preliminary. The detailed comparison of contemporary geospatial processing systems is a topic for future work. In their 2018 paper, Mauri et al. [47] performed benchmarks on Magellan[10], GeoSpark, GeoTrellis and PostGIS. Investigating the performance of spatial join operations, the authors conclude that "*there was no intersection query where Apache Spark libraries outperformed PostGIS*". When comparing the performance of PostGIS and MongoDB for spatial-temporal data, Makris et al. [46] found that "*PostgreSQL outperforms MongoDB in all cases and queries*". Also, "*the dataset size occupied in the system db, reduced 4x in case of PostgreSQL*". In his 2016 thesis, M. Toups

[10]We disregarded Magellan, as it did not support the latest versions of Apache Spark.

[68] compared PostGIS with GeoMesa and a flat-file architecture called Vector Cluster. He found that "*GeoMesa query times [...] are very certainly significantly worse than the Vector Cluster and PostGIS times*". And, "*One surprising result was the very fast performance of PostGIS/PostgreSQL, which easily out-performed Vector Cluster on all queries except the few with the largest payloads*".

Based on those findings, we decided to use PostGIS for integrating external data sources with our patch dataset.

External Data Sources

We have selected two different types of information for integration with our patch data: Land cover and country labels.

Land cover information is obtained from the CORINE Land Cover inventory (Bossard et al., 2000 [12]). In it's latest version, *CLC 2018*, 44 classes document land cover properties in 39 participating countries at better than 100m positional accuracy. Included attributes are: Shape polygons, shape statistics (length of the shape, area covered) and land cover labels. The data is available in both raster and vector formats. A detailed list of land cover classes can be found in the Appendix C.1.2.

Information about which country a patch is located in is obtained from the NUTS dataset (Eurostat, 1995 [26]). Developed by the European Union, the *Nomenclature of Territorial Units for Statistics* is comprised of three distinct levels: NUTS 1 describes "*major socio-economic regions*", NUTS 2 "*basic regions for the application of regional policies*" and NUTS 3 "*small regions for specific diagnoses*". The latest 2016 version of the dataset includes the 28 EU member countries, 5 EU candidate countries and 4 EFTA countries. All regions covered by CORINE, with the exception of Kosovo, are also covered by NUTS. We will use shape polygons from NUTS 1 to obtain the English country names for our image patches.

Job Design

Before we can start annotation process, we first have to obtain a copy of the CORINE and NUTS datasets and upload it into the PostGIS database.

Once executed, the job's parallelization logic added three empty columns to the image patch `DataFrame`: One for the country labels, one for the Land cover label and one for the CORINE *object IDs*, used as a reference to the associated CORINE shapes. Then, a UDF is applied to every partition of the `DataFrame`. For every partition, one SQL connection is acquired. Spatial intersection queries are executed sequentially for each partition element against the CORINE and NUTS tables. The results are first added to the empty columns and then collected by the parallelization logic. Patches for which no labels were found were excluded. Fully annotated patches were then written to a user-specified parquet file. A mechanism for inserting the patch metadata into an SQL table was also provided.

Once the annotation in complete, further analysis, e.g. modification of patches with a specific land cover class, can be performed using PySpark's query processor. To avoid unnecessary I/O overhead, we strongly recommend to use the SQL backend for analysis tasks that do not require access to the sensor data.

4. Implementation

4.1. Overview

In this chapter we describe how we implemented our prototype for an end-to-end framework for processing and analysis of big data in earth observation. Following the design specified in Chapter 3, the implementation is divided into three parts: Data acquisition, processing and analysis.

The first section covers the CollGS parallel download tool. We describe the structure of the implementation, what software was used to implement it and how to install, configure and deploy it. The implementation of our PySpark workflow concept is the topic of the second section. We describe in detail what steps were taken in building the distributed atmospheric correction, patch generation and data export jobs. Also, we explain job configuration, installation and deployment to the processing cluster. In addition to this, we describe the processing cluster configuration, job unit testing and how executor log files were aggregated. The third section focuses on analysis of the generated image patch dataset. Building on the strategies introduced in the previous section, we describe how we used both internal and external data sources to obtain scene classification, land cover and country labels for the patches.

4.2. Data Acquisition

The CollGS prototype is written in the Python programming language (Van Rossum et al., 1995 [69]) and compatible to Python v3.6 and upwards. It is licensed under the GNU General Public License Version 3 and available for download under

`https://gitlab.tu-berlin.de:rsim/CollGS.git`

Incremental versioning is done following PEP 440 (Coghlan & Stufft, 2013 [20]). Where ever possible the use of packages from the Python standard library was preferred over those from third parties. In its latest version, `0.1.4`, the following third party packages are required:

- `sentinelsat` [61]
  Interface to Copernicus OpenSearch / OpenData API endpoints
- `PyYAML` [76]
  For read / write access to a YAML configuration file
- `pytest` [55]
  For convenient unit testing
- `pyarrow` [54]
  For interfacing with Hadoop Distributed File System

Optionally, *CollGS* API documentation can be generated using the `pdoc` [52] package.

4.2.1. Package Structure

Following the de facto standard for organizing Python packages described in K. Reitz's essay *Repository Structure and Python* [57], the *CollGS* prototype is structured as follows:

- `collgs/`
  Python module containing business logic
  - `__init__.py`
    Marks this folder as a Python module
  - `cli.py`
    Code for a command line interface
  - `manager.py`
    Code for the `SentinelAPIManager` class; Concurrent mirror access, download scheduling etc.
  - `utils.py`
    Utility functions used in the two other files
- `docs/`
  Contains automatically generated documentation
- `tests/`
  Contains all Unit tests
- `config.yaml`
  Configuration file template
- `LICENSE`
  A copy of the GNU General Public License Version 3
- `README.md`
  A markdown formatted Readme file
- `requirements.txt`
  A list of required Python dependencies, following PEP 508 [21] directives
- `setup.py`
  For convenient installation of the package, including references to all dependencies

4.2.2. Installation

The Python programming language's standard library provides extensive support for building and installing modules through the `setuptools` package. Package metadata, including the name of the package, its current version, its author(s) and the employed license are specified in the `setup.py` file located in the package's root directory. For the CollGS package, a list of package dependencies, additional file includes and details for a line entry point were also added.

Leveraging the `setuptools` workflow results in the following user experience: After obtaining the CollGS source code from the version control system users execute the *setup.py* file to install the package to an existing Python installation. Dependency resolution is automatically performed on installation by the `pip` package management system. Moreover, the `setup.py` file allows users to create binary packages as well as execute unit tests. The command line entrypoint created by `setuptools` allows users to invoke the CollGS command line interface just like any other binary on the system path.

In combination with Python's *virtual environment* feature, which help users to avoid installing Python packages globally, the chosen approach allows a convenient installation of the package with minimal requirements and access rights.

4.2.3. Configuration

Configuration parameters can be passed to the `SentinelAPIManager` class constructor either via keyword arguments or by passing the path of a configuration file. Both methods are allowed to be used simultaneously, covering a variety of usage scenarios. Parameters passed to the command line interface are routed to the constructor as keyword arguments. YAML was chosen as the data serialization language of choice due to it's minimal syntax and high readability. Configuration files are parsed into a key-value mapping in the class constructor. Python's `dict` type is also used to represent keyword arguments. Before being assigned as attributes to the class instance the provided configuration parameters are first merged into a single `dict` object and then validated for correctness. Missing key-value pairs are replaced by default arguments on parameter validation. As keyword arguments offer a slightly more dynamic configuration approach than static configuration files parameters passed via keyword arguments overwrite those read from a configuration file, allowing for ad-hoc changes to existing configurations read from file. In the following I will list the configuration parameters currently supported by `SentinelAPIManager`:

- `base`
  Path were downloaded files are stored
- `cloud`
  Max. allowed cloud cover percentage $[0.0, 1.0]$
- `connections`
  Max. allowed connections per server
- `date`
  Products must be acquired in the specified range
  - `from`
    Exclude products acquired before this date (YYYY-MM-DD)
  - `to`
    Exclude products acquired after this date (YYYY-MM-DD)
- `hdfs`
  HDFS master node details
  - `host`
    Master node URL
  - `port`
    Master node port
  - `user`
    Master node user name
  - `base`
    HDFS base directory
- `mirrors`
  OpenSearch/OData API endpoint details
  - `url`
    Endpoint URL
  - `user`
    Endpoint user name
  - `password`
    Endpoint user password
- `parallel`
  Max. allowed total connections

- `retry`
  Max. allowed number of retries per mirror
- `test`
  List of UTM zones the performance score is calculated from
- `timeout`
  Connection read / connection timeout limits (seconds)

4.2.4. Command line interface

Two routes are offered to invoke the *SentinelAPIManager* class: Either directly, by calling the object constructor or indirectly, by using the command line interface.

Following the example of the `sentinelsat` package, the CollGS prototype emphasizes the role of the command line as the preferred interface for remote sensing workflows. Albeit both routes have feature parity, the command line interface, which is based on the Python standard library `argparse`, has the additional benefit of providing even inexperienced users with fast and easy access to a number of common usage patterns through so called *subcommands*. These subcommands, specified as the second argument of a call, abstract the most commonly used sequences of `SentinelAPIManager` and auxiliary function calls. Furthermore, they allow exposing a number of subcommand-specific configuration parameters to the command line. The command line interface can thus be understood as a *third* logical layer, based on `SentinelAPI` and `SentinelAPIManager`, that focuses on the end user. The following subcommands are available in *CollGS* v0.1.4:

- `collgs search`
  Acquire metadata for an area of interest
- `collgs select`
  Select product with lowest cloud cover per MGRS zone and season
- `collgs describe`
  Calculate product statistics
- `collgs preview`
  Download preview images
- `collgs get`
  Download and decompress products
- `collgs list`
  List existing files
- `collgs mirror`
  Interact with configured mirrors
  - `list`
    List all configured mirrors
  - `rank`
    Calculate mirror performance score
- `collgs hdfs`
  Interact with Hadoop Distributed File System
  - `list`
    List products on HDFS
  - `upload`
    Upload products to HDFS
  - `download`
    Download products from HDFS

All of the above commands require a *target* to be specified as the first argument. Supported inputs are: MGRS tile ids (comma separated string or CSV file) and GeoJSON footprints as well as JSON files following the structure defined in Section 3.2.7 (i.e. result of previous `collgs search` or `collgs select` operations).

With the exception of multiple login details (see `mirror` in Section 4.2.3) and UTM testing zones (see `test` in Section 4.2.3) *all* configuration parameters available in the configuration file can also be specified to the command line interface (using the `--` prefix). If both are present, options passed to the command line interface will overwrite those specified in the configuration file. The following subcommand-independent configuration parameters can be specified to the `collgs` command line interface:

- `--user`
  Endpoint user name
- `--url`
  Endpoint URL
- `--password`
  Endpoint user password
- `--timout`
  Connection read / connection timeout limits (seconds)
- `--retry`
  Max. allowed number of retries
- `--parallel`
  Max. allowed total connections
- `--sentinel`
  Filter by Sentinel mission number $[1,2,3]$
- `--producttype`
  Filter by Sentinel product type
- `--base`
  Path were files are stored
- `--config`
  Path to configuration file (default: `config.yaml`)
- `--verbose`
  Activate verbose logging

Further information regarding usage of the `collgs` toolchain is shown on specification of the `--help` option.

4.2.5. Utility functions

The `utils.py` includes a number of custom Python functions that are used by, but not part of the `SentinelAPIManager` implementation. Among those are functions for

- Reading and writing specific file formats (JSON, YAML, CSV, ZIP)
- Detecting and validating data formats (UTM zone identifier, SAFE file format)
- Extracting information from data returned by `SentinelAPIManager` (Season of acquisition, product UUIDs, product UTM zone ids)

4.2.6. Unit Tests

Python Unit tests are implemented with the `pytest` framework, which provides a variety of powerful concepts for testing Python source code. Separate tests for all three parts of the `collgs` module (`cli.py`, `manager.py` and `utils.py`) are provided in the `tests/` folder. Integration into the open-source DevOps solution GitLab (Hethey, 2013 [34]) allowed us to continuously test new versions of the CollGS toolchain on deployment to the Version Control System. Commits with failing tests were rejected.

4.3. Processing

The processing stage of our end-to-end framework prototype is comprised of three *jobs*:

1. Generation of Level-2A products from Level-1C products (Section 4.3.4)

2. Generation of fixed-size patches from Level-2A images (Section 4.3.5)
3. After successful analysis, export of annotated patches (Section 4.3.6)

All processing jobs presented here were implemented in Python v2.7[1], using PySpark v2.4.2 bindings. The job code is included in our exemplary processing and analysis workflow, which can be obtained from

`https://gitlab.tubit.tu-berlin.de/rsim/BigEarthNet2`

As for the `CollGS` downloader, the workflow is licensed under the GNU General Public License Version 3. Incremental versioning is done following PEP 440 (Coghlan & Stufft, 2013 [20]). The workflow is structured according to the layout proposed in Section 3.3.2.

Before we move on to describe the job implementation in more detail, we first elaborate on more general additions made to the processing system.

4.3.1. Build & Submit Utilities

The build and submit utilities, defined in Section 3.3.3 and 3.3.4, were implemented in `bash` script. Both utilities are essentially collections of command line calls, modularized to work with all jobs organized according to the layout introduced in Section 3.3.2. While the build script utilizes the `conda` command line interface to generate the virtual environment, the job submit script is built on `spark-submit`. We tested our framework with conda v4.6.* and spark-submit v2.4.*.

Both tools follow the same approach: First, they use a user-specified job name to identify the job directory. Then, the associated files are identified. Finally, the respective calls are executed. To distinguish different job versions from each other, the version number can be specified in the header of the scripts.

Build Script

The build script creates a new Python v2.7 conda environment, named after the job name and version. In case an environment with the same name already exists, no new environment is created, but the existing one updated. Once this is done, it locates conda `env.yml` and pip `requirements.txt` requirements files. Both locally, in the job directory and globally, in the scripts working directory. The requirements are installed using the respective package manager. Global requirements are installed first. Any Python module located inside the job folder is then copied to the the environment's Python module container, `.../lib/python2.7/site-packages/`. They same is done for the content of the `dependencies/` folder, in case one is located in the script working directory. Finally, `conda pack` is used to create a environment `.tar.gz` file in the `envs/` folder. Existing tarballs with same job name / version combination are overwritten.

Submit Script

The submit script identifies four types of files to be submitted to the cluster: The `job.py` job script, the environment tarball, the `config.yml` configuration file and any JAR file placed in a `jars/`

[1]We chose this soon to be deprecated version of the Python interpreter as it was required by the employed atmospheric processor, Sen2Cor. Without access to the patent protected parts of it's codebase, we were unable to to conduct a migration to the latest version of the Python interpreter ourselves. To ensure compatibility of processing jobs among each other, Python v2.7 was used for all of them.

folder located either in the job folder or the script working directory. The first two are mandatory for the submission of a new job. Using the obtained information, the `spark-submit` call is built and executed.

In addition to the identified files, a number of cluster manager settings, defined in the header of the script, are passed to `spark-submit`. In the following, all options used in the submit script are listed:

- `--master`
  What cluster manager to use, default `yarn`
- `--deploy-mode`
  Whether to start the driver process on a worker node (`cluster`) or locally (`client`), default `client`
- `--archive`
  Archives to be extracted to the YARN container root, used for deploying virtual environment with dependencies
- `--files`
  Files to be distributed to the executors, used for distributing the `config.yml` file
- `--jars`
  Jar files to be distributed and added to the executors Java path, used for deploying PostgreSQL JDBC bindings
- `--num-executors`
  The number of executors to be spawned by the cluster manager, default 69
- `--executor-cores`
  The number of cores per executor, default 3
- `--executor-memory`
  The amount of RAM per executor, default 28 GB
- `--config`
  Additional Spark options
  - `spark.yarn.appMasterEnv.PYSPARK_PYTHON`
    Set the `PYSPARK_PYTHON` environmental variable on the YARN Application Manager, point it to the Python binary provided by the virtual environment
  - `spark.executorEnv.GDAL_DATA`
    Set the executors' `GDAL_DATA` environmental variable, point it to the path of the GDAL library inside the virtual environment

The `master` and `deploy-mode` parameter can also be specified as second and third argument to the `submit.sh`, which will overwrite the default value.

4.3.2. Cluster Configuration

The two systems employed by us to realize the processing jobs, Apache Spark and Hadoop, both accept a range of configuration parameters. In the previous section, we have documented what options were specified to Apache Spark on job submission. Here, we will further elaborate on what default values were used in our test setup, and why we chose them.

Executor Resources

Based on the hardware specifications of our test cluster, described in Section 2.8, the default values for the Spark executor resource configuration were calculated as follows:

- Based on previous experiences with the cluster hardware, we chose **3 cores per executor**, to allow for optimal I/O throughput
- With 3 out of the 24 CPUs on each node reserved for system processes, 21 CPUs are available per node
- This means 7 executors per node
- We have 9 connected worker nodes, hence **69 executors in total**
- Leaving 50 GB of the 250 GB RAM per worker for system processes, 200 GB of RAM are available per node
- Divided by 7 executors per node, a bit more than **28 GB of RAM** are available per executor

YARN

Aside from our personal resource manager configuration the following options where specified to the Hadoop YARN configuration:

- `yarn.log-aggregation-enable` = `true`
  Activates aggregation of executor log files.
- `yarn.nodemanager.vmem-pmem-ratio` = `5.1`
  Ratio between virtual and physical memory for container allocation. Increased to allow submitting large environment tarballs.
- `yarn.nodemanager.resource.memory-mb` = 204800
  Physical memory per node that can be allocated for containers. Adjusted for our hardware setup.
- `yarn.scheduler.maximum-allocation-mb` = 204800
  Maximum memory to be allocated per container. Adjusted for our hardware setup.
- `yarn.nodemanager.resource.cpu-vcores` = 22
  Maximum number of CPUs to be allocated per container. Adjusted for our hardware setup.

4.3.3. Logging

Log files are a vital tool for gathering run-time information of processing jobs. Error logs help identify problems with the job logic, while the standard output can be used to report internal debug and performance information. However, aggregating log files from distributed PySpark jobs can be cumbersome for a number of reasons: Although there is a central logging interface, Spark provides no reference to the original context to the executors, so executor logs can not be forwarded to the main logger. To further complicate things, cluster managers, like YARN, spawn the executors inside of virtual *containers*, which are immediately destroyed after job completion. Without modifying the respective cluster manager, copying container log files or re-routing the executors' `stdout` / `stderr` streams is close to impossible.

Fortunately, the YARN cluster manager provides a feature for aggregating executor logs on the HDFS. To activate it, we have to modify both the YARN and Spark configuration: First, we need to set the `yarn.log-aggregation-enable` parameter in the `yarn-site.xml` file to `true`. Then, we specify the default log file directory, we used a folder inside the `/tmp` directory, to the Spark configuration parameter `spark.yarn.app.container.log.dir`. Once this is done, we configure the Python `logging` module to write the executor logs to a custom log file inside this folder. The logfiles can then easily be accessed via the Hadoop Web interface or the `yarn logs` command line interface.

A convenience method for obtaining the executor logger object was added to the global workflow dependencies.

4.3.4. Distributed Sen2Cor

The atmospheric correction job was based on Sen2Cor v2.8, which is available for download under

```
http://step.esa.int/main/third-party-plugins-2/sen2cor/sen2cor_v2-8/
```

As described in Section 3.3.5, a number of problems prevent us from including an unmodified version of Sen2Cor in a PySpark job:

1. The package is distributed as a standalone installer.
2. Package dependencies are largely unknown.
3. A number of additional files need to be distributed.
4. Parts of the code are only available as pre-compiled binaries, dynamically linked against libraries provided in the installer.

Extracting the Source Code

The first challenge was addressed by writing a small shell script for extracting the Sen2Cor Python module from the standalone installer. On execution of the installer a self-contained Python 2 environment is extracted to disk. Similar to the virtual environments discussed above, this environment contains *all* files required to run Sen2Cor, including a copy of the Python interpreter. In theory, this environment could already be used in a PySpark job. We decided against this option, as it would mean deploying nested environments while surrendering control over the dependencies. Also, adding modifications to the code would be considerably harder. Instead, we chose to further isolate the Sen2Cor processor from the environment it is shipped in.

As with any other Python distribution, installed modules can be found in the `site-packages` directory. To extract the Sen2Cor source code, it was copied from this directory.

A detailed listing of the Sen2Cor package structure can be found in the Software User Manual [30] on page 11.

Discovery of Dependencies

Two dependencies are explicitly named in the Sen2Cor user manual [30]: `GDAL` and `PyTables`. However, to successfully start the processor, more dependencies were required. In order to identify them, an initial list of candidate dependencies was compiled from the names of the modules that were present in the `site-packages` folder. It was merged with a list of modules that are imported in the readable source files. For every candidate, the official Python Package Index (PyPI) was searched for similar or matching package names. After deduplication, the resulting list of packages formed the initial requirement file for the isolated Sen2Cor module. The respective dependency versions were determined with the following approach:

1. Beginning with the latest version of the package
2. It was tried to install and run the package (using Python's `setuptools`)
3. In case the above step failed, the error logs were searched for the package(s) that caused the error and the respective version number decreased
4. If no errors were encountered, the process was stopped

With the help of the above strategy we were able to identify 29 dependencies required for running Sen2Cor outside of the environment provided by the installer. A detailed list of package names and versions can be found in the Appendix B.1.1.

Distribution of Additional Files

After detecting the dependencies we focused on the third problem: Distributing auxiliary files required by Sen2Cor. By default, Python's `setuptools` includes only Python source files when building a package. However, for the Sen2Cor processor to work correctly, some auxiliary files also need to be included. After consulting the manual we identified four sources of such files:

1. `aux_data/`
   Containing a global snow map (1 File, 24 MB)
2. `cfg/`
   Containing a number processor configurations and file specifications (440 Files, 15 MB)
3. `lib_S2A/`
   Containing Sentinel-2A specific Look-up-Tables for radiative transfer calculations used in the atmospheric correction (240 files, 49 MB)
4. `lib_S2B/`
   Containing Sentinel-2B specific Look-up-Tables for radiative transfer calculations used in the atmospheric correction (242 files, 49 MB)

The above sources were included in the isolated Python package by adding them to a `Manifest.in` file, `setuptools` default way for modifying the include path. By specifying a recursive load instruction, we made sure all files inside the source directories were included.

Handling Binarized Executables

As described above, Sen2Cor's atmospheric correction stage, contributed by the German Aerospace Center, is copyright protected and provided as a binary. To be included in our isolated Sen2Cor package, it had to be specified in the `MANIFEST.in` file. To further complicate things, the respective binary was dynamically linked against a version of the `musl libc`, an implementation of the C standard library, which was located in a different folder of the environment provided by the installer. Removing the binary from it's original path also meant breaking the link. Simply copying the particular library to our package did not solve this problem either, as the library path was hardcoded in the ELF header. Recompiling the source was out of the question. A manual configuration of the `LD_LIBRARY_PATH` on every worker node seemed overly complicated. We thus chose to add a *run-time search path*, called `rpath` in ELF terminology, to the binary header. To do so, we used the `patchelf` [50] utility provided by the NixOS Linux distribution. Using the `rpath` allowed us to specify a custom location for the `musl libc` and hence solved the issue of handling binarized executables in our isolated Sen2Cor package. Finally, a copy of the `musl libc` was added to the package using the `Manifest.in` file.

Modifying Sen2Cor

The isolated, original version of Sen2Cor was further modified to allow processing of files located on the HDFS. We initially tried to use the JNI-based HDFS interfaces, like the one provided by the `pyarrow` [54] library used in `CollGS`, but were unable to correctly set it up on the executors. Alternative WebHDFS endpoints were not available at our setup. As Hadoop binaries were by default installed to any worker node connected to the YARN cluster manager, we then devised a simple Python interface for abstracting calls to `hdfs` command line interface. Products located on the distributed filesystem were first copied to a temporary directory on the executors' file system before applying the atmospheric processor. Successfully converted Level-2A products were then moved back to the HDFS and the remote path returned.

Even though licensed under the Open Source Apache 2 license, public redistribution of Sen2Cor source by a third party is explicitly forbidden (see Page 43, Sen2Cor User Manual [30]). Hence, we will not provide a link to our modified packages here, but attach them to the code submitted alongside this thesis.

For further usage in the PySpark job, the modified processor module is compiled into a *Python wheel*, a package format that allows direct installation via the `pip` package manager.

Job Implementation

After successfully extracting and modifying the Sen2Cor processor, we move on to implementing the PySpark job. Following the layout introduced in Section 3.3.2, the distributed atmospheric correction job is structured as follows:

- `atcor/`
  The PySpark job container
  - `config.yml`
    A YAML file containing the job's configuration parameters
  - `sen2cor-hdfs.whl`
    A Python package (”*wheel*”) containing the modified Sen2Cor processor

  - `job.py`
    The job's initialization logic tasked with loading input data, validating config parameters and storing results
  - `requirements.txt`
    Contains the job's requirements. For this job, only a reference to the Sen2Cor wheel is specified
  - `S2AtCor/`
    Module containing parallelization logic
    * `__init__.py`
      Indicates that this folder is a Python module
    * `atcor.py`
      Contains the parallelization logic, invoked by `job.py`, calls the Sen2Cor processor

The `job.py` script contains three functions: One `main` method, one for validating the configuration parameters and one for uploading the result data to a SQL database. On submission to the cluster manager, the `main` method starts the Spark context, loads the `config.yml` file and validates the passed parameters. Missing values are replaced and wrong ones reported. Then, the input data is read from a JSON file and converted to a PySpark `DataFrame`, using a custom `json_to_sql` utility functions which is located in the workflow's `dependency` module. The imported dataset of Level-1C product paths and respective product UUIDs is passed to the `parallel_sen2cor` method defined in `S2AtCor.atcor`. Here, a `sen2cor_wrapper` UDF, the job's business logic, is mapped against all partitions of the input data. We chose partitions to be the subject of the map algebra (`mapPartitions` in PySpark RDD API) and not individual partition elements (`map`) in order to reduce the overhead caused by initialization of the dependencies. The UDF iterates over all elements of the partition, calls the `L2A_Process.main` method of the modified Sen2Cor processor and yields a new `Row` object containing the path to the converted Level-2A product, a new, ISO11578 [38] compliant UUID and a reference to the original L1C product.

Failed products are retried for a user-specified number of times. If the retry limit is exceeded, a row with empty Level-2A product path and UUID is returned. On completion of the `mapPartitions` transformation, the `parallel_sen2cor` methods returns a `DataFrame` of `Row` objects to the `job.py` script. Using the output file path set in the configuration file, the `DataFrame` is written to disk. In case PostgreSQL login details were specified, the initialization logic tries to upload the result data to an SQL table. Job duration and number of succeeded / failed conversion are logged before the initialization logic terminates the PySpark context.

Column	**Type**
`l1c_uuid`	`str`
`l1c_path`	`str`
`l2a_uuid`	`str`
`l2a_path`	`str`

Table 4.1. *`DataFrame` structure returned by `parallel_sen2cor`*

Installation & Deployment

The *build & submit utilities*, introduced in Section 3.3.3 and 3.3.4,
allow convenient generation of the dependency archive and submission to the cluster manager.

On calling

```
./build.sh atcor
```

the modified Sen2Cor processor and the `S2AtCor` module, containing the PySpark parallelization logic, are first installed to a `conda` environment and then compressed to a tarball. The Sen2Cor wheel, specified in the requirements file, is installed by the `pip` package manager. On detecting the `__init__.py` file, the build script copies the `S2AtCor` module to the environment's `site-packages` folder.

To start the job, the initialization logic must be submitted to the cluster manager. On calling

```
./submit.sh atcor
```

the `job.py`, alongside the `config.yml`, the environment tarball and a number of cluster settings, are passed to the `spark-submit` script. Depending on the specified cluster options, `spark-submit` either directly spawns the Spark driver process (deploy mode `client`) or returns the result of the job submission (deploy mode`cluster`).

Configuration

A number of configuration parameters for the distributed atmospheric correction job are exposed in the `config.yml` file:

- `input`
  Information about the input data
  - `mode : string`
    Where to load the L1C data from, currently only `json` is supported
  - `file : string`
    Location of the JSON file, if mode is `json`
- `output`
  Information about the output data
  - `file : string`
    Where to store the output DataFrame
  - `partitions : integer`
    How many partitions the output data should have
- `postgres`
  PostgreSQL database login details, optional
  - `host : string`
    Host name
  - `port : integer`
    Host port
  - `user : string`
    User name
  - `password : string`
    User password
  - `database : string`
    Database name

Metadata Management

The above schema was registered with a PostgreSQL database. Before starting the atmospheric correction job we used our custom `json_to_sql` utility function to insert the Level-1C product information from the CollGS JSON file into the SQL database. Once the processing in completed, the initialization logic tries updating the `l2a_products` table. An SQL command to create the above tables is included in `sql/create_tables.sql`.

l1c_products		
Attribute	**Type**	**Note**
uuid	UUID	primary key
path	TEXT	L1C path
info	JSONB	OpenSearch JSON

l2a_products		
Attribute	**Type**	**Note**
uuid	UUID	primary key
path	TEXT	L2A path
l1c_product	UUID	Foreign key

Table 4.2.
SQL schema for storing metadata from the distributed atmospheric correction job.

4.3.5. Patch Generation

Following the design specifications described in Section 3.3.6, we divide the patch generation process into three distinct *transformations* which we implemented as methods of a `Sentinel2PatchGenerator` class:

1. `get_band_infos`
   Extract sensor band file infos from Level2A SAFE paths

2. `load_patches`
   Load sensor data and generate image patches

3. `merge_patches`
   Aggregate single-band patches into multi-band ones

To simplify usage of the `Sentinel2PatchGenerator`, a forth `generate_patches` method combines all calls required to a generate a complete patch dataset into one.

Internally, the class methods were all structured similarly: First, we define the business logic as a nested function. While it is generally recommenced to use the runtime-optimized SparkSQL algebra to do this, none of the tasks at hand could be fully described with its limited feature set. Instead, we resorted to non-optimized UDFs. By declaring them as nested functions we make sure that only the respective UDF, and not the whole class instance, is copied to the executors[2]. Also, this structure allows us to use the arguments passed to the `Sentinel2PatchGenerator` inside of the business logic UDFs, as if they were declared as global variables. Once defined, we use the PySpark map algebra to apply the UDF to the specified input `DataFrame`. The transformed `DataFrame` is then returned to the initialization logic.

The `Sentine2PatchGenerator` class attributes are used to store information used by more than one of its methods, e.g. a reference to the PySpark logging object.

In order to be recognized by the `build.sh` as a job dependency, the `Sentine2PatchGenerator` class is placed in a Python module that we've called `S2PatchGen`.

[2]When defining UDFs as class methods, PySpark tries to serialize and copy the whole class instance to the executors. Depending on the structure of the class, this could create unwanted overhead.

Initialization logic and configuration file are added, resulting in the following job structure:

- `patchgen/`
  The PySpark job container
  - `config.yml`
    A YAML file containing the job's configuration parameters
  - `job.py`
    The job's initialization logic tasked with loading input data, validating config parameters and storing results
  - `requirements.txt`
    Contains the job's requirements. For this job, references to the NumPy, pytest and GDAL packages are specified
  - `S2PatchGen/`
    Module containing parallelization logic
    - `__init__.py`
      Indicates that this folder is a Python module
    - `generator.py`
      Parallization and business logic are defined here, inside the `Sentine2PatchGenerator` class. Invoked by `job.py`.

The following dependencies are defined in the requirements file: All access to geospatial data is provided by v2.3.3 of the GDAL library (Warmerdam, 2008 [72]). For unit testing we used pytest v4.6.5 [55] and mock v3.0.5.

We will now describe the steps taken to generate the image patches in more detail.

Retrieving Sensor Band Information

Before we can load the sensor data, we first have to identify the location of the sensor band files inside the L2A SAFE directory. This is achieved in the `get_band_infos` method by applying a user-defined function to every element of the input `DataFrame`.

Using the L2A product paths generated by the distributed atmospheric correction job, we first locate the so-called *image granules* - Standardized directories inside the Sentinel-2 products containing the sensor data. To simplify testing and debugging of the transformation, both HDFS and non-distributed file system paths are supported. After locating the image granules, the relevant sensor band files are identified. We define two criteria on the basis of which we classify files as relevant for further processing:

1. The name of the band is **not** included in a user-specified list of band names to be excluded from processing.

2. The band resolution is the same as in the original Level-1C product, thus excluding bands resampled by Sen2Cor.

Using a regular expression the band name, band resolution and MGRS zone identifier are extracted from the selected band file paths. The combined band information is returned as an array of `Row` types.

The arrangement of the retrieved band information is of decisive importance for the following transformation, the loading of the sensor data. As all bands of a returned `Row` object are read at once, the more bands a `Row` contains, the more resources are required by the respective Spark executor to load them. Depending on the dimensioning of the executors, the resource requirements may exceed the supply, leading to idle times and degraded system performance.

To accommodate for this we offer two *modes* for retrieving the band information: `wide`, which will return one band per `Row` and `narrow`, which will include all bands selected from a product per `Row`. The mode argument is specified to the `get_band_infos` method. While the first option helps us to avoid excessive resource consumption when loading the sensor data, it also has a disadvantage: We want our final image patches to match the multi-spectral nature of the input products and hence contain data from *multiple* bands. This requires grouping the data read from single bands somewhere along the processing chain. Aggregation operations like this can however lead to an excessive amount of *shuffles*, i.e. redistribution of data across partitions, which is bound by the Disk and Network I/O and hence a costly operation. Under such circumstances, using the `narrow` mode can become viable.

Column	**Type**
`l1c_uuid`	`str`
`l2a_uuid`	`str`
`l2a_path`	`str`
`bands`	`array`
`bands.band_name`	`str`
`bands.band_path`	`str`
`bands.resolution`	`int`
`bands.utm`	`str`

Table 4.3.
`DataFrame` structure returned by `get_band_infos`

Regardless of the chosen mode, the returned `Row` objects are joined with the input data to form the output `DataFrame` described in Table 4.3.

Loading the Sensor Bands & Generating Patches

Loading of the sensor data from the retrieved band information as well as the generation of image patches is handled by the `load_patches` method. Because of the large amounts of data handled in this transformation, we had to take particular care to maintain a small memory footprint. In addition to the arrangement of the input data, discussed in the previous subsection, this goal is achieved mainly through the use of two particular programming techniques:

1. Instead of loading the entire content of a band to memory before passing it to the patch generation mechanism, we only load a *reference* to the band data. The patch generator then selectively loads only that data which is required for the respective patch from the reference.

2. By using the lazily evaluated Python *generator* functions the patch generation strategy returns patches *on demand*. Only when there is capacity to load new patches, they are generated. The memory footprint is significantly smaller as there is not need to store all results in memory before returning them.

In the `load_patches` method, the `gdal_load_and_split` business logic UDF is applied via a `flatMap` operation to each `Row` of the input data, a `DataFrame` with band information generated by the `get_band_infos` method. In comparison to Spark's one-to-one `map` operation, a `flatMap` can yield zero, one or more elements. This follows the ratio that from one tile, we will create many patches.

Inside the UDF, all band files associated with the respective `Row` are copied from the distributed file system to the local executor context. Then, GDAL is used to create a reference to the band data (`Dataset.GetRasterBand`) and extract the band size (`Dataset.Raster*Size`), bounding coordinates (`Dataset.GetGeoTransform`) and spatial reference system identifier (`Dataset.GetProjection`). The combined data is passed to the `make_patches` generator, where in a two-dimensional for-loop a non-overlapping sliding window is applied to each band. What we will later refer to as *patch index* are the number of window shifts in X and Y direction.

The window size is determined by the pixel resolution: For 10m accuracy $120 \times 120 pixel$ are extracted, for 20m $60 \times 60 pixel$ and for 60m $20 \times 20 pixel$. Whatever portion of the band not covered by a full window size is ignored. The pixel values are selectively read from the band data reference into a `numpy` array using GDAL's `Dataset.ReadAsArray` method. With the original band coordinates, new patch bounding-box coordinates (upper-left and lower-right corner) are calculated. Then, we prepare the final image patch structure:

For each band name, path, resolution, size, band data and pixel value statistics (minimum, maximum, mean) are added to a `Row`. As, at the time of writing, Apache Spark did not support multi-dimensional arrays, the patch data has to be flattened into a one-dimensional PySpark `Vector` first before it can be added. The resulting `Row` objects are added to a Python dictionary, whichs maps the band name to the respective band data `Row`. This *band mapping* is then added to another `Row` object were all patch-related data is collected. Besides the band data this includes: Patch coordinates, patch index, patch UUID, spatial reference system identifier, MGRS zone identifier and references to the original L1C and L2A products. The final patch `Row` objects are then returned by the `make_patches` generator and collected by the `gdal_load_and_split` UDF. Before they are registered with the transformed `DataFrame`, the references to the band files are closed and the files removed from the local executor context. Also, the duration and the number of generated patches is logged.

Column	**Type**
`uuid`	`str`
`l1c_uuid`	`str`
`l2a_uuid`	`str`
`lrx`	`float`
`lry`	`float`
`ulx`	`float`
`uly`	`float`
`epsg`	`int`
`utm`	`str`
`xidx`	`int`
`yidx`	`int`
`bands`	`map`
`bands.band_name`	`str`
`bands.band_path`	`str`
`bands.data`	`Vector`
`bands.width`	`int`
`bands.height`	`int`
`bands.min`	`float`
`bands.max`	`float`
`bands.mean`	`int`
`bands.xres`	`int`
`bands.yres`	`int`

Table 4.4.
`DataFrame` structure returned by `load_patches`

Merging Image Patches

When the `wide` mode option was used to retrieve the band information, the above patch generation stage is applied to single bands only. The resulting band map therefore only has one entry. In the `merge_patches` transformation, such single-band patches are merged into multi-band ones. To achieve this, we apply the PySpark aggregation function `combineByKey` to the patch dataset. First, each `Row` of the input `DataFrame` is transformed into a key-value pair. The key is generated by combining the L2A UUID and the patch index. The value is the input `Row`. On this key-value data, we apply the aggregation operation.

`combineByKey` requires three input arguments: A function that defines how *combiner* objects are created, a function that defines how new rows are merged with combiner objects and a function that defines how to merge combiner objects with each other. What we refer to as combiner objects are the data structures that will be used for aggregation - In our case, the patch `Row` structure created in the previous transformation. For all partitions, Spark will generate as many combiner objects as there are distinct keys present in the partition. Whenever a matching combiner objects is found in their respective partition, the remaining partition elements are merged with those combiner objects. In our case, we update the combiner band mapping with the data obtained from the single-band patch. Finally, all of a partition's combiner object keys are broadcasted to the other partitions. Combiner objects with the same key are merged according to the instructions specified in the third function passed to `combineByKey`. Again, we simply merged one combiner band map into another here.

Once all patches are aggregated, we reverse the key-value structure and return the multi-band patches in a `DataFrame`. Depending on the structure of the job script, the data is then either passed on to further analysis or written to a user-defined parquet file.

Job Implementation

The `job.py` script contains two functions: One `main` function and one for validating the configuration parameters. On submission to the cluster manager, the `main` method starts the Spark context, loads the `config.yml` file and validates the passed parameters. Missing values are replaced and wrong ones reported. Then, the input data, which is expected to be structured like the output of the atmospheric correction job, is read from a parquet file specified in the job configuration. An instance of the `Sentinel2PatchGenerator` class is created and it's `generate_patches` method called to start the patch generation process. Along with this call a number of parameters, read from the configuration file, are passed to `generate_patches`, which assigns them to their respective transformations. Also, user can control the number of partitions between transformations. Once completed, the `DataFrame` containing the generated patches is written to a user specified parquet file.

Installation & Deployment

We will again make use of the job management capabilities provided by the *build and submit* workflow. On calling

```
./build.sh patchgen
```

the `S2PatchGen` module, along with the libraries specified in the requirements file, are installed to a `conda` environment and compressed to a tarball.

To start the job, the initialization logic must be submitted to the cluster manager. On calling

```
./submit.sh patchgen
```

the `job.py`, alongside the `config.yml` and the environment tarball are passed to the `spark-submit` script. Depending on the options specified to the submit script, `spark-submit` interface either directly spawns the Spark driver process (deploy mode `client`) or returns the result of the job submission (deploy mode`cluster`).

Note: Since workflow version 1.0 the `generate_patches` method includes additional calls to the analyis methods described below. In order to generate unlabeled patches only, references to these methods must be removed.

Configuration

The configuration file for the patch generation job exposes the following options:

- `input`
  Information about the input data
  - `mode : string`
    Where to load the L1C data from, currently, only `parquet` is supported
  - `file : string`
    Location of the parquet file
  - `partitions : integer`
    How many partitions the input data should have
- `output : string`
  Path to the output file
- `generator`
  Configuration for the patch generator
  - `mode : string`
    How band information are organized, `wide` or `narrow`
  - `exclude : array`
    What band names to exclude from processing
  - `partitions : int`
    How many partitions the patch data should be organized in

Metadata Management

The metadata from the generated patch `DataFrame` can easily be uploaded to an SQL database using JDBC. After setting up the appropriate login details calling

```
df.drop("bands").write.jdbc(url=jdbc_url, table="patches",
                            properties=jdbc_props)
```

will write all patch metadata to a `patches` table. Except for the missing band information, the structure of the created table will be equivalent to Table 4.4.

4.3.6. Data Export

In addition to the tools provided by the build & submit workflow, the distributed generation of a GeoTIFF archive will mainly be based on GDAL v2.3.3 (Warmerdam, 2008 [72]). A Python implementation of the JSON (Bray, 2014 [13]) file format is used to write metadata files. For compressing the partition data we use the ZIP format (Katz, 1989 [53]), which is provided by the Python standard library `zipfile`.

Similar to the atmospheric correction job, the parallelization logic is organized in a single function, `export_geotiff`. When invoked, it applies the `dump_zipfile` UDF via a `mapPartitionsWithIndex` transformation to all partitions of the patch data. Again, we make use of Python generator functions in order to minimize the memory footprint. As indicated by the name, the `mapPartitionsWithIndex` transformation forwards an additional partition index to the UDF, which will be used in naming the ZIP files.

The dump_zipfile UDF first acquires a GeoTIFF *driver* from GDAL (GetDriverByName("GTiff")), which in the following will be used to create the images. It then opens the ZIP file that the partition data that will be used to compress the partition data. As the compression backend we choose the default zlib (Gally & Adler, 2017 [28]).

Once all the prerequisites are set up, the export process is started. For all partition elements the *patch directory* names are generated according to the design specification described in Section 3.3.7. A regular expression is used to extract the satellite mission, acquisition date and time from the band path. The remaining information are obtained from the patch Row. Then, the patch directory is created in the executor working directory and the metadata exported to a JSON file. Missing scene, land cover and country labels are ignored. For all bands of the partition elements the GeoTIFF driver is used to create a new file (Dataset.Create). The geocoordinates are set (Dataset.SetGeoTransform) and the coordinate reference system is configured (Dataset.SetProjection). Finally the band data, reshaped back into a two-dimensional NumPy matrix, is added (Dataset.Raster.WriteArray

The generated files are then copied to the ZIP file. Once all partition elements have been exported, the non-compressed data is deleted from the executors. The ZIP file is then moved to the HDFS and a Row, containing the new HDFS path, is returned to the parallelization logic. export_geotiff converts the returned objects into a DataFrame and returns them to the job script.

Structure

The distributed data export job is structured as follows:

- export/
  The PySpark job container
  - config.yml
    A YAML file containing the job's configuration parameters
  - job.py
    The job's initialization logic tasked with loading input data, validating config parameters and decompressing the dataset
  - requirements.txt
    Contains the job's requirements. For this job, references to the NumPy, pytest and GDAL packages are specified
  - S2PatchX/
    Module containing parallelization logic
    * __init__.py
      Indicates that this folder is a Python module
    * exporter.py
      Parallization and business logic are defined here. Invoked by job.py.

The following dependencies are defined in the requirements file: GDAL v2.3.3, NumPy 1.16.5 and pytest v4.6.5.

The job script contains four functions: In addition to the ones used in the other jobs, main and validate_config, we defined a method that obtains compressed partitions from the HDFS and decompresses them to a local directory. Also, we added a Python *context manager* to facilitate multi-threaded decompression with the help of Python's multiprocessing module. On submission to the cluster manager, the `main` method starts the Spark context, loads the `config.yml` file and validates the passed parameters. Then, loading of the patch data from a user-specified parquet file is scheduled and the `export_geotiff` transformation executed. The results are collected by the Spark driver process in a Python `list`. Using the context manager, we spawn a thread pool with as many threads as the driver node has CPUs. For every ZIP file contained in the results, a thread downloads and decompresses the data to a user-specified directory. Once completed, the local ZIP files are removed and the number of decompressed files, as well as the duration, are reported.

Installation & Deployment

Installation and deployment of the distributed data export job is handled by the *build and submit* workflow. On calling

```
./build.sh export
```

the `S2PatchX` module, along with the libraries specified in the requirements file, are installed to a `conda` environment and compressed to a tarball.

To start the job, the initialization logic must be submitted to the cluster manager. On calling

```
./submit.sh export
```

the `job.py`, alongside the `config.yml` and the environment tarball are passed to the `spark-submit` script. Depending on the options specified to the submit script, `spark-submit` interface either directly spawns the Spark driver process (deploy mode `client`) or returns the result of the job submission (deploy mode`cluster`).

Configuration

The following configuration parameters are exposed in `config.yml` file:

- `input : string`
  Path to the input parquet file
- `output`
  Information about the output
  - `zips : string`
    Driver node directory were ZIP files are downloaded to
  - `extract : string`
    Driver node directory were ZIP files are decompressed to
- `keep_zips : boolean`
  If True, do not delete ZIP files from the driver node

4.3.7. Unit Tests

Both the patch generation and export stage are covered with `pytest` [55] unit tests. A Spark instance is started locally to allow testing without connecting to a remote cluster. With the help of an example band file, the structure and content of the data returned by the PySpark transformations is verified.

`Sentinel2PatchGenerator` methods are tested one after the other. Successfully tested methods are used as input for the next ones to test. Due to the nested design of the transformations, utility methods defined inside the business logic had to be re-implemented in the test script. Calls to the HDFS are mocked to simulate a working distributed file system.

No tests for the distributed atmospheric correction stage are included, because reaching an adequate level of test coverage for the Sen2Cor module would have exceeded our limited time frame. The Sen2Cor authors were asked about this topic, but until the end of this thesis we did not receive an answer.

Similar to the test setup for CollGS, we enabled continuous testing of the processing code through GitLab (Hethey, 2013 [34]). Commits that cause tests to fail are not merged into the remote repository.

4.4. Analysis

The analysis stage of our end-to-end framework is implemented on top of the processing stage. All modifications made to the patch generation mechanism are included in our exemplary processing and analysis workflow, which can be obtained from

`https://gitlab.tubit.tu-berlin.de/rsim/BigEarthNet2`

Two methods were added to the `Sentinel2PatchGenerator` class:

- `scene_label_patches`
  Annotate patches with scene classification labels
- `corine_label_patches`
  Annotate patches with land cover and country label information

In the following, we will describe the additions to the patch generator in more detail.

4.4.1. Scene Classification Label Annotation

In the `scene_label_patches` method, scene class labels are extracted from the scene classification map band created by Sen2Cor. The extraction logic is defined in the `find_scene_labels` UDF. We apply the UDF to the patch data via the PySpark `withColumn` transformation, which also creates a new `DataFrame` column from the results. Inside the UDF, a mapping of scene mask pixel values (integer) to label string, based on the values presented in Figure 3.7, is stored. Using this mapping, a distinct set of scene class values is extracted from the patch's SCL band and converted into the corresponding strings. The resulting array of strings is returned to the parallelization logic and automatically added to the new `scene_label` column.

In case the `drop_nodata` option is specified to the transformation, all patches containing a 0, the value for the `NO_DATA` class, in their SCL bands are filtered from the `DataFrame` before generating the scene labels.

4.4.2. Land Cover & Country Label Annotation

In the `corine_label_patches` method, land cover and country labels are integrated with the patch dataset by using the PostGIS spatial database. Spark executors are connected with PostGIS using `pyscopg2` (Varrazzo, 2019 [70]) a PostgreSQL client for Python. Before we can start the process we first have to set up at least one database instance and insert the CORINE and NUTS datasets.

In our test setup, we chose to install the database on the largest node available to us, a machine with 50 cores and 500 GB RAM. PostgreSQL v10.10 was obtained from the Ubuntu APT package repository. Version 2.5.3 of PostGIS was built from source, acquired from

```
https://download.osgeo.org/postgis/source/postgis-2.5.3.tar.gz
```

An ESRI Geodatabase formated (`.gdb`) copy of the 2018 version of the CORINE dataset, CLC 2018, was downloaded from

```
https://land.copernicus.eu/pan-european/corine-land-cover/clc2018
```

The NUTS 2016 dataset, again in ESRI Geodatabase (`.gdb`) format, was fetched from

```
https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/
administrative-units-statistical-units/nuts
```

The GDB files were then inserted to the PostGIS instance using GDAL's `ogr2ogr` feature conversion tool. The following call was used to insert the NUTS table:

```
ogr2ogr -f "PostgreSQL" PG:"host=$host port=$port dbname=$db user=$user
password=$password" ref-nuts-2016-01m.gdb -overwrite -progress --config
PG_USE_COPY YES
```

The same command can be used to insert the CORINE data, only the name of the input file has to be changed. As the original CLC 18 data does not include human readable land cover labels, but only a label *code*, we had to insert an additional table, mapping the codes to readable label strings. An SQL dump of that table is provided as `sql/labels.sql` with our example workflow.

When all tables are inserted, the `corine_label_patches` transformation can be started. First, three new array-type columns (`corine_labels`, `corine_objectids`, `country`) are added to the patch `DataFrame`. Following this, the `corine_label_partitions` UDF is applied via the `mapPartitionsWithIndex` operation. In the UDF we try to establish a connection to the PostGIS instance. If this fails we retry for a user-defined number of times. Once a connection is established we iterate over all partition elements and execute the SQL queries against the PostGIS instance. The request to the CORINE table (see Figure 4.1) is executed first.

The PostGIS `ST_MakeEnvelope` function is used to generate a *geometry* type from the patch bounding-box coordinates. With the `ST_Transform` command the geometry is transformed to the ETRS89 coordinates reference system used by the CLC shapes (SRID number: 3035). Then, the `ST_Intersects` command is used to find spatial intersection between the CORINE objects and the respective patch. The results are joined with the `labels` table to obtain readable class labels. Finally, the distinct set of CORINE object IDs and land cover labels is returned to the UDF. There, the values are added to the newly created `DataFrame` columns. Patches for which no data was found are filtered out from the dataset.

After fetching the land cover labels, the query to obtain the country label is executed (see Figure 4.2).

An analysis of the PostgreSQL query plan confirmed that dynamically generating the patch geometry with `ST_MakeEnvelope` and `ST_Transform` significantly outperforms referencing pre-computed

```sql
SELECT DISTINCT clc.objectid, labels.label
FROM clc2018_clc2018_v2018_20 clc, labels
WHERE ST_Intersects(clc.shape, /* compute intersection */
  ST_Transform(                /* transform to CLC SRID */
    ST_MakeEnvelope(           /* make patch bounding box */
      $ulx, $uly, $lrx, $lry, $epsg
    ),
  3035)
) AND clc.code_18 = labels.code;
```

Figure 4.1.
SQL query to obtain CORINE land cover label

```sql
SELECT DISTINCT name_engl
FROM nuts_rg_01m_2016_3035 nuts
WHERE ST_Intersects(nuts.shape, /* compute intersection */
  ST_Transform(                 /* transform to NUTS SRID */
    ST_MakeEnvelope(            /* make patch bounding box */
      $ulx, $uly, $lrx, $lry, $epsg
    ),
  3035)
```

Figure 4.2.
SQL query to obtain NUTS country label

geometries, hence we again use this approach in the NUTS query to generate the patch bounding box. Like the CLC dataset, NUTS uses the ETRS89 coordinates reference system. Spatial intersection with the NUTS shapes are computed with the `ST_Intersects` function. A distinct set of English country names is return to the UDF and added to the respective patch `Row`. This time patches for which no country labels were found are included in the dataset[3].

After obtaining labels for all elements of a partition the PostGIS connection is closed. Duration and number of successfully labeled patches are logged and the updated partition returned to `corine_label_patches`.

4.4.3. Installation & Deployment

The analysis was implemented as part of the `Sentinel2PatchGenerator` class. With workflow version 1.0 we updated the `generate_patches` wrapper function to include calls the analyis methods. Therefore, the installation and deployment process of the analysis stage is equivalent to that described in Section 4.3.5.

4.4.4. Configuration

The configuration file of the `patchgen` job was extended with the following parameter:

[3]To accommodate for the fact that Kosovo is not included in NUTS and that parts of the ocean will not be associated with any country.

- `filter`
  Information about what patches to exclude from the annotated dataset
  - `drop_nodata : boolean`
    Whether or not to exclude patches with `NO_DATA` scene labels
- `postgis`
  PostGIS database login details
  - `host : string`
    Host name
  - `port : integer`
    Host port
  - `user : string`
    User name
  - `password : string`
    User password
  - `database : string`
    Database name

4.4.5. Metadata Management

patches		
Attribute	**Type**	**Note**
`uuid`	`UUID`	primary key
`l2a_product`	`UUID`	foreign key
`bbox`	`GEOMETRY`	bounding box
`corine_labels`	`TEXT[]`	land cover
`corine_objectids`	`INT4[]`	foreign keys
`country`	`TEXT[]`	country
`scene_labels`	`TEXT[]`	scene classes

Table 4.5.
SQL schema for storing metadata from the distributed analysis job.

Building on the technique described in Section 4.3.5 the insertion of annotated patch metadata was implemented in the `Sentinel2PatchGenerator.update_database` method. Using the PostgreSQL JDBC connector the patch `DataFrame` (without band data) is inserted to a temporary table. The content of the temporary table is then converted to the layout described in Table 4.5. Spatial reference information are converted into a PostGIS `geometry` types. An SQL command to create the above table is included in `sql/create_tables.sql`.

4.4.6. Unit Tests

As in the processing stage the analysis code is covered with `pytest` unit tests. Tests for the `Sentinel2PatchGenerato` class were extended with test for the `scene_label_patches` and `corine_label_patches` transformations. Additionally, a number of tests for PostGIS were added: Transformations between coordinate reference systems were tested for a loss of accuracy. Intersection queries for the NUTS and CORINE tables were tested against a list of ground-truth values.

5. Evaluation

In this section we evaluate the performance of our end-to-end framework. We examine each stage of our implementation, data acquisition, processing and analysis at three different scales:

1. **Small-scale**: With three images from the Berlin area (`0.9 GB`)
2. **Medium-scale**: With 352 images from Germany (`171 GB`)
3. **Large-scale**: With 2824 images from 38 European countries (`1.4 TB`)

For the small-scale test case we selected images from one particular grid zone - The Berlin area (MGRS: `33UUU`), home of TU Berlin. We ran it on a single worker node of our cluster to create an initial performance baseline measurement.

For the medium-scale test case we used a GeoJSON representation of the German border to programmatically identify 84 MGRS zones that contain data for this area. From the one-year period of July 2017-2018 we then selected 352 Sentinel-2 L1C products with zero percent cloud coverage, one for each season, so up to 4 for each grid zone[1]. Note: Since the data for this test was downloaded a few months before the others, some of the selected parameters differ.

For the large-scale test case the Sentinel-2 tiling grid [1] was used to identify 866 MGRS grid zones from 38 states participating in the latest CORINE Land Cover inventory - 28 EU member countries, 5 EU member candidates, 4 EFTA states and Kosovo. From the one-year period of July 2018-2019 we selected 2824 products, again one for each season.

We now proceed with a detailed evaluation of the individual stages and then conclude the chapter with an evaluation of the end-to-end performance.

5.1. Data Acquisition

We used our CollGS data acquisition tool to identify and acquire suitable data for the above test cases. Four collaborative ground segment accounts were added to the configuration: SciHub (ESA), Finland, Norway and Greece. All other mirrors were either found to have disabled their API endpoints (Portugal, Sweden), changed API specifications (Germany), showed high error rates (Austria, Italy) or left the Collaborative Ground Segment program (UK).

5.1.1. Product Search

The `collgs search` subcommand was used to perform an initial search in the L1C product catalogs. For the Berlin area grid `33UUU`, 13 cloud-free products were identified in the July 2018-2019 period. On average a search was completed in 1.36 seconds (n=5). For our medium-scale test case, 1013

[1]Due to a bug in our selection mechanism, seasons of the same name from different years were not considered to be the same. Thus we selected more than four products for some zones.

products were identified in 23.25 seconds. In contrast to the other two cases, products with up to ten percent cloud coverage were included. For the large scale case 13048 cloud products were found in 1285.63 seconds. For 42 of 866 specified grid zones no cloud-free products were found. Search results were stored in a JSON file.

5.1.2. Product Selection

Next, we used the `collgs select` subcommand to narrow down our previous selection to one product per grid zone and season of the year. All three operations were completed in less than a second. Results were exported to a JSON file. The following number of products was selected: 3 for small-scale, 352 for medium-scale and 2824 large-scale.

The selections statistics shown in Table 5.1 were generated with the help of the `collgs describe` subcommand. Additional season statistics can be found in Appendix D.1.

Scale	**Products**	**Avg. Size**	**Total Size**	**Avg. Cloud**
small	3	313.61 MB	0.91 GB	0.0%
medium	352	499.17 MB	171.58 GB	6.9%
large	2824	509.33 MB	1404.63 GB	0.0%

Table 5.1.
Description of the evaluation datasets after selecting one product per season and MGRS zone.

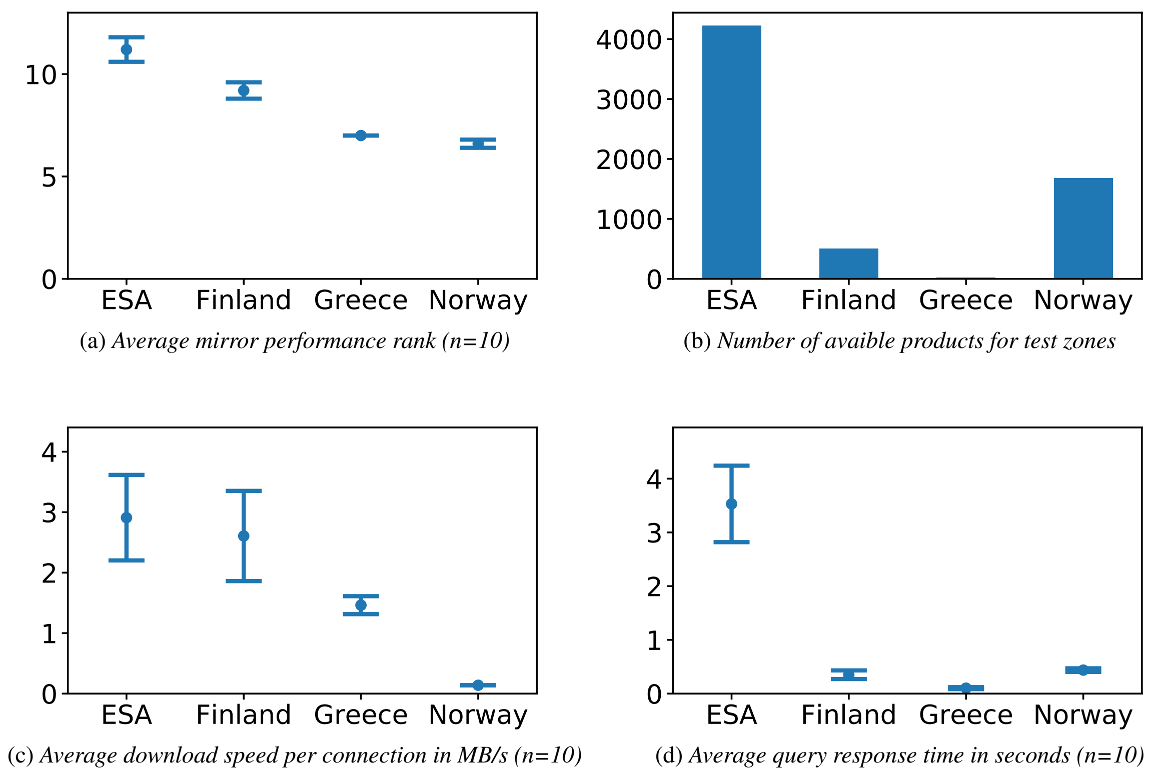
5.1.3. Mirror Ranking

Before we started the data acquisition process we first evaluated the performance of the configured mirrors using the `collgs mirror rank` subcommand. We selected 7 MGRS zones, one from each European Collaborative Ground Segment member, for collecting the performance metrics: `33UWP` (Austria), `35VLG` (Finland), `32UQD` (Germany), `34SGH` (Greece), `31UES` (France), `32VNM` (Norway), `32TQM` (Italy). Allowed cloud cover percentage was set to zero percent. To accommodate for varying connection quality we repeated the ranking process ten times and calculated the average values. The results are shown in Figure 5.1.

The highest average performance value, 11.2 ± 0.6, was measured for ESA's SciHub mirror, followed by Finland with 9.2 ± 0.5, Greece with 7.0 ± 0.0 and Norway with 6.6 ± 0.2. With 4229 available products the ESA mirror also outperforms all other mirrors with regards to the total number of products available for the given test zones. The Norwegian mirror came in second with 1679 files, followed by Finland with 507 and Greece with 19.

With one exception, the results of the speed measurement were similar to the spring 2019 measurements shown in Table 3.1. In terms of average download speed the SciHub mirror, with 2.9 ± 0.7 MB/s, performs only slightly better than Finnish one, with 2.6 ± 0.75 MB/s. While not as fast as the first two, the Greek mirror, with 1.46 ± 0.14 MB/s, exhibits a relatively low standard deviation. In contrast to the earlier measurement however, the speed of the Norwegian Mirror seemed to have been massively reduced: From 48.5 ± 3.2 MB/s in January 2019 down to 0.14 ± 0.0 MB/s in September 2019.

The average request duration was calculated by sending a `GET` request to the mirrors' OpenSearch endpoints and measuring the time it took the server to respond. For all three national mirrors average response times of under one second were measured. The Greek mirror answered the fastest, with 0.10 ± 0.01 seconds, followed by the Finnish one, with 0.35 ± 0.08 seconds, and the Norwegian one



(a) *Average mirror performance rank (n=10)*

(b) *Number of avaible products for test zones*

(c) *Average download speed per connection in MB/s (n=10)*

(d) *Average query response time in seconds (n=10)*

Figure 5.1.
Evaluation of the mirror performance ranking.

with 0.44 ± 0.03 seconds. The server with both the largest data archive and the largest user base, ESA's SciHub mirror, was also the slowest to respond with an average of 3.53 ± 0.71 seconds.

5.1.4. Product Download

After completing the mirror ranking we started the data acquisition with the `collgs get` subcommand. A maximum of 6 parallel connections were allowed, with 2 connections per server. The timeout was set to 15 seconds and the number of retries limited to 50. To compensate for the loss of our two most promising Mirrors, Germany (changed API specs) and Norway (heavily reduced download speed), we used a trick to bypass the SciHub connection and rate limit: By adding another set of SciHub user login details we were able to raise the number of available connections by two to a total of four. The available bandwidth increased accordingly. On receiving the fifth connection from the same IP address however, the SciHub mirror blocks all connections from this address for the next 2 hours[2]

The results of the acquisition process, as shown in Table 5.2, show the positive impacts of the concurrent mirror access strategy on the download speed. Average speeds up to five times greater than the fastest individual connections were achieved. The test data for the small-scale was acquired in 2.07 minutes, for the medium-scale test in 3.46 hours and for the large-scale test in 26.1 hours.

[2]While this successfully demonstrates the capabilities of the concurrent mirror access strategy, multi-account use is often seen as rude behavior. The bandwidth limitations exist for a reason, don't be rude.

Scale	Size	Duration	Avg. Speed
small	0.91 GB	2.07 min	7.5 MB/s
medium	171.58 GB	3.46 h	14.1 MB/s
large	1.37 TB	26.1 h	14.9 MB/s

(a) *Download duration and speed*

ESA	Finland	Greece	Norway
100%	0.0%	0.0%	0.0%
86.9%	13.0%	0.0%	0.0%
91.2%	8.2%	0.5%	0.1%

(b) *Percentage of downloads per mirror*

Table 5.2.
Evaluation of the data acquisition results

Another visible feature is SciHub's dominance in the distribution of downloads per mirror. In knowledge of our scheduling strategy this can be attributed to the following factors: As the highest scoring one the SciHub mirror is preferred. It also has the largest amount of products available (see Figure 5.1) and thus all of the 4 available connections will be in use most of the time. As we have limited the amount of total connections to 6 this leaves only 2 connections to the other mirrors. Whether or not those can then occupy the remaining connections depends on the size of their product catalogs. But the product portfolios offered by national mirrors are usually limited to the borders of the operator nation, which further skews the distribution towards the SciHub mirror. In addition, the smaller mirrors were slower so that fewer downloads could be completed overall.

In the two longer running download sessions our fault tolerance strategy could also demonstrate its usefulness. Six read timeouts were reported during the medium-scale test. The system successfully recovered by retrying the respective downloads. In the large-scale test 67 retries were reported. When the SciHub server was taken offline for maintenance shortly before the session was completed, the retry limit was quickly exceeded and the 52 remaining downloads were marked as failed. Due to time constraints, we continued with the reduced data set.

5.1.5. HDFS Upload

After finishing the downloads we uploaded the files to the HDFS using the `collgs hdfs upload` subcommand. The updated product paths were exported to a JSON file. For all three cases we achieved stable upload rates of around 55 MB/s. It took 16.9 seconds to upload the small-scale, 53.1 minutes to upload the medium-scale and 7.1 hours to upload the large-scale test data.

5.2. Processing & Analysis

After the L1C products were uploaded to the HDFS we started the processing stage. With the help of our build & submit workflow job environments were packaged and submitted to the cluster (see Section 2.8). Configuration parameters were set in the job's `config.yml` files.

5.2.1. Atmospheric Processing

Paths to the JSON files returned by the HDFS uploader were specified to `jobs/atcor/config.yml`. The number of partitions for the output file was set to the respective number of executors. We decided not to retry failed conversions because in all the cases we observed the problem was with the HDFS and not with the Sen2Cor logic. This particular issue, which lead to some executor not being able to write to the HDFS, continued until the end of this stage. Before starting the next stage, however, we were able to identify a wrongly mounted partition on one of the storage nodes and thereby fixed the problem. As an additional metric for comparing the scales, we calculated the average number of products processed per minute.

In a newly installed Anaconda distribution building the job environment took 6.2 minutes. A subsequent run utilized the pre-cached packages and was completed in 3.2 minutes.

Scale	L2A	Duration	Avg. Sen2Cor	L2A/min
small	3	41.7 min	29.2 ± 3.5 min	0.07
medium	341	5.4 h	28.4 ± 7.1 min	1.05
large	2443	37.6 h	31.5 ± 6.1 min	1.08

Table 5.3.
Evaluation of the atmospheric correction results

As shown in Table 5.3, the atmospheric correction for the small scale tests was completed in 41.7 minutes with three Spark executors. This corresponds to a throughput of 0.07 products per minute. From the aggregated log file the average duration per conversion was calculated to be 29.2 ± 3.5 minutes, indicating that a significant portion (12.5 ± 3.5 minutes) of the job was spent on setting up and pulling down the Spark context.

The medium-scale test was completed in 5.4 hours using all 69 scheduled executors. With 28.4 ± 7.1 minutes the average duration per conversion was comparable to that of the the small-scale test. Due to problems with the HDFS 11 out of 352 (3.1%) conversion jobs failed, reducing the number of generated L2A files to 341. This corresponds to a throughput of 1.05 products per minute.

The large-scale test was completed in 26.1 hours again using all 69 Spark executors. The average duration per conversion was again very similar to the previous test with 31.5 ± 6.1 minutes. 381 out of 2824 product conversions failed which amounts to a failure rate of 13.5%. With 2443 successfully converted products a throughput of 1.08 products per minute is achieved. This is the highest value of the three tests. Especially the pronounced difference to the small-scale test highlights the benefits of distributing the processing tasks. Medium- and large-scale tests perform similarly, suggesting that the maximum throughput depends mainly on the amount of resources allocated.

On completion of the processing the L2A metadata was uploaded into an SQL table. The upload was completed in 0.5 seconds for the small-scale, 1.6 seconds for the medium-scale and 12.6 seconds for the large-scale experiment.

5.2.2. Patch Generation

After successfully completing the atmospheric correction we set up the configuration file for the patch generation job. Parquet files created by the previous stage were specified as input. Band information were read in `wide` mode. Since the `narrow` mode repeatedly led to memory-related crashes of the executors, no comparison could be made between the two modes. The TCI, AOT and WVP bands were excluded from loading. A PostGIS database instance was configured as described in Section 4.4.2. The analysis stage was configured to filter out all patches containing the `NO_DATA` scene classification label. System throughput was measured in number of L2A products processed per hour.

Figure 5.2.
Example image patch, Sweden, Pastures & Water courses

The time required to build the job environment was very similar to that of the previous job: Packaging in a newly installed Anaconda

distribution was completed in 6.5 minutes. A subsequent run took
3.4 minutes.

Scale	Patches	Duration	L2A/h
small	16413	24.43 min	6.72
medium	1.71 M	19.63 h	17.37
large	10.3 M	150.87 h	16.19

(a) *Patch generation statistics*

Info	Load	Scene	PostGIS
0.12%	6.3%	0.49%	41.95%
0.14%	10.2%	0.52%	87.42%
0.17%	10.19%	0.66%	87.49%

(b) *Percentage duration of subtasks*

Table 5.4.
Explanation of the column names in (b): Info includes loading of band information, Load includes loading and merging of patches, Scene includes annotation with scene classification labels, PostGIS includes integration of labels from PostGIS and creation of a patch dataframe.

Table 5.4 shows the result of the patch generation stage. In the small-scale test, 16413 patches were generated in 24.43 minutes. As 3 input files were processes this amounts to a throughput rate of 6.72 products per hours. Identification of the band files was completed in 1.77 seconds and produced 39 bands. Before loading the bands the `DataFrame` was repartitioned from 3 to 150 partitions. Loading and merging of the patches took 1.54 minutes and produced 24843 patches, 8281 patches for each product. Annotation with scene classification labels was completed in 7.24 seconds. 8340 (33.57%) of the patches contained `NO_DATA` labels and where removed. A total of 10.25 minutes (41.95%) was spent on integrating data from PostGIS. The largest share of time, 12.49 minutes (51.13%), was not spent in any of the subtasks and can be attributed to setup and pulldown of the Spark environment. For all patches both the land cover and country labels were found. The generated parquet file had a size of 1.1 GB.

For the medium scale test 1.710.348 patches were generated in 19.63 hours. 341 input files were processed, yielding a throughput rate of 17.37 products per hour. Identification of the band files was completed in 2.04 minutes and produced 4433 bands. After obtaining the band infos the data was repartitioned from 69 to 2500 partitions. Loading and merging of the patch data was completed in 2.55 hours and produced 2823821 patches. Integration of scene labels took 7.80 minutes and lead to the exclusion of 1113473 (39.43%) `NO_DATA` patches. By far the largest share of time was spent with annotation of country and land cover labels, 17.61 hours, which amounts to 87.42% of the total job duration. Again, we were able to find both label types for all specified patches. 18.58 minutes (1.58%) of the time could not be attributed to any subtask. The patch data parquet file had a size of 221.3 GB.

For the large-scale test 10.301.669 patches were generated in 150.87 hours. Considering 2443 files were specified as input we calculated a throughput rate of 16.19 products per hour. Identification of 31759 sensor bands was achieved in 15.0 minutes. Before loading the patches the `DataFrame` was repartitioned from 69 to 15000 partitions. Loading and merging of 20.230.483 patches took 17.2 hours. Annotation of scene class labels was completed in 1.0 hours. 7.079.998 (35.0%) of those patches were removed because of a `NO_DATA` tag. Integration of PostGIS data was achieved in 132.0 hours. For 2848816 patches (21.66% of the remaining patches), no CORINE labels could be found. As a result those patches were not included in the final patch archive. After reviewing the metadata for some of the rejected patches we identified two main causes: Either they were located in international waterways or countries that did not participate in the CORINE Land Cover inventory. Similar to the previous task 25.25 minutes (0.28%) of the time spent could not be attributed to any subtask. The parquet file containing the patch data had a size of 1.3 TB.

A number of trends become apparent: Integration of labels from the PostGIS database is by far the most time consuming of all subtasks. The percentage for the large and medium test is almost equal. Also, the PostGIS throughput rate stayed very similar: 1601.27 patches/min for the small-scale, 1618.73 patches/min for the medium-scale and 1660.41 patches/min for the large-scale experiment. Around 1600-1700 patches/minute seems to be the maximum throughput rate for the selected PostGIS setup regardless of the number of connections used. Our preliminary hypothesis is that the number of CPUs available for the PostGIS processor, which remained the same in all three tests, is the decisive factor.

While we seem to have reached the maximum system throughput rate in the medium-scale test, the difference to the large-scale test is rather small. As with the atmospheric correction stage, this leads us to conclude that not the size of the input data but the amount of assigned resources are the decisive factor for the overall system performance.

Although the percentage time share for identifying sensor band information and integrating scene class labels is always below 1%, their share increases with the size of the data set. A similar dynamic can also be observed for loading and merging of the patches. Contrary to what was feared, however, we could not observe any shuffle-induced exponential growth of the time share for this subtask. On the contrary, with 19603.18 patches/minute the patch load throughput rate for the large-scale test was even slightly faster than that of the middle-scale test (18456.34 patches/minute).

After completing the patches generation the patch metadata was uploaded to a PostGIS database using the provided `update_database` function. The upload for the small-scale test data was completed in 7.2 seconds, for the medium scale in 14.71 minutes and for the large-scale in 33.41 minutes. As an example for the usefulness of the database we calculated the number of patches without country labels for the large-scale test, 1896615, in less than a second.

5.2.3. Data Export

Once all patches were generated we continued to export them to a GeoTIFF archive. Parquet files generated by the patch generation stage were configured as input files. For storing the zipped partitions we specified a path on the HDFS. For extracting the archive we allocated a total of 3.5 TB of space on the most powerful machine in our cluster (50 CPUs). There we also launched the Spark driver process. ZIP files obtained from the HDFS were configured to be deleted after extraction.

Scale	Duration
small	26.05 min
medium	4.91 h
large	34.7 h

Table 5.5.
Export job durations

For the small-scale test 16413 patches in 150 partitions were exported in 26.05 minutes. ZIP files were generated in 7.1 minutes (27.25%) and extracted in 9.3 minutes (35.7%). 9.65 minutes (37.04%) were spent outside of the two subtasks. The final archive had a size of 3 GB.

For the medium-scale test 1.71 million patches in 2500 partitions were exported in 4.91 hours. GeoTIFF partitions were generated in 18 minutes (6.1%) and extracted in 4.4 hours (89.61%). Similar to the previous experiments 12.6 minutes (4.29%) were spent outside of any subtask. The extracted dataset had a size of 320 GB.

For the large-scale test 10.3 million patches in 14062 partitions were exported in 34.7 hours. While generating the GeoTIFF ZIP files was completed in 2.65 (7.63%) hours, download and extraction of the ZIP files amounted to 31.78 (91.58%) hours. 16.2 minutes (0.79%) of the time was spent outside of the two subtasks. The extracted archive had a size of 1.9 TB.

As clearly visible from the results, the export job is split into a distributed (Generation of the partition ZIP files) and a non-distributed (extraction if ZIP files) part. While the generation of the

ZIP files benefits from the fact that it is executed on several machines in parallel, the extraction process is subject to a loss in performance, which is typical for working large volumes of data on a single machine. In all three test cases both CPU and RAM usage for the extraction process was relatively low, indicating that the hard disks were the cause of this bottleneck. However, it is also unclear to what extent the chosen compression algorithm contributes to this effect.

5.3. End-to-End

After completing all stages of our end-to-end framework we compared the overall system performance for the three selected scales. As an indicator for system performance we again calculated the throughput rate, this time relative to the volume of data processed.

Scale	Input	Duration	GB/h
small	0.9 GB	1.60 h	0.57
medium	171.58 GB	38.80 h	4.42
large	1.4 TB	249.27 h	5.99

(a) *End-to-end statistics*

DL	AtCor	PatchGen	Export
2.16%	43.44%	27.92%	27.14%
8.92%	13.92%	64.51%	12.65%
10.47%	15.08%	60.52%	13.92%

(b) *Percentage duration of stages*

Table 5.6.
Explanation of the column names in (b): DL includes data acquisition, AtCor includes atmospheric correction, PatchGen includes patch generation and analyis, Export includes exporting of GeoTIFF archives

For the small-scale test the end-to-end run was completed in 1.6 hours. As documented in the previous sections a significant part of this is contributed by system setup and pulldown overhead, which highlights the disadvantages of running smaller batches in the presented framework.

For the medium-scale test one complete run took 38.8 hours. In comparison to the small-scale tests the importance of achieving a high level of parallelism became clear.

For the large-scale test the four sections of the framework were completed in 249.27 hours. Throughout all conducted experiments the performance of the large-scale test was comparable to that of the medium-scale case, which is again shown by the time share percentage presented in Table 5.6. Other than expected, the size of the dataset did not have a negative effect on the system performance. On the contrary, the overall throughput rate of 5.99 GB/s was even greater than that of the medium-scale test with 4.42 GB/h. Although one must consider that a significant proportion of the large-scale test data either contained `NO_DATA` values or no CORINE labels and was thus not passed to the subsequent stages. The comparison of the two scales on the basis of the processed input data is therefore only of limited value for understanding the overall system performance.

6. Conclusion and Discussion

In the past chapters we have designed, implemented and evaluated an end-to-end framework for processing and analysis of big data in remote sensing. We began with a system for the fast acquisition of satellite data. On the basis of an existing API client we developed a strategy for concurrently accessing multiple open-access mirrors for Copernicus data, the so called Collaborative Ground Segments. In combination with a custom download scheduling mechanism we were able to achieve a five-fold increase in download speed compared to existing, non concurrent approaches. This gain would probably have been even greater if several ground segment operators had not changed or even disabled their services. Especially worth mentioning are the German and Norwegian mirrors which were each more than three times as fast in our preliminary examination as the maximum value reached at the end. While the German mirror replaced the standardized Ground Segment API specification with a custom one at the beginning of 2019, the Norwegian mirror reduced its download speed by more than fifty times of the original value. The loss of these two largest ground segments ultimately led to the fact that most of the increase in speed achieved was due to concurrent access to the ESA-run SciHub server. Although our system could also demonstrate its usefulness under these more difficult conditions, the question arises whether the effort to develop a complex parallel system was justified. By building on an open-source client with an active community we were able to outsource much of the API-specific implementation work, but the effort required to manage concurrency was still considerable. While commercial solutions for the acquisition of satellite data, in contrast to the approach presented here, are usually subject to a fee, they offer significantly faster connections, which renders the use of concurrent approaches unnecessary. Such providers also offer several year old data, which is usually removed from the product catalog on open-access servers. Regardless of the achieved download rates, the approach developed by us is tailored specifically to the requirements of a big data remote sensing task, as we were able to demonstrate in several experiments. The search, selection and acquisition of several thousand Sentinel-2 products was achieved within a few calls to the provided command line interface. The built-in failure tolerance mechanism provides effective protection against short-term connection problems, preventing premature termination of long-running download sessions. Another helpful feature was the built-in distributed file system interface which allowed acquired products to be quickly uploaded to the HDFS.

For the processing and analysis of the acquired earth observation data we developed a system based on two well-known big data tools: Apache Hadoop and Apache Spark. Looking back, we conclude that this was a good choice - Both are mature systems with a large and active user base. To address their missing support for remote sensing tasks we first developed a number of tools with which we could extend the functionality accordingly. Based on the standard Python package managers the presented design allows the convenient extension of Apache Spark by arbitrary Python libraries via the Hadoop YARN cluster manager. By reducing the underlying logic to two basic commands, build and submit, we were able to massively reduce the effort required to generate and launch such modified Spark jobs. This is particularly useful in the present case as it allows users with little prior knowledge about big data technologies, e.g remote sensing experts, to use the system. Based on the job templates defined in this context we subsequently developed a series of Spark jobs with which we successfully generated a large-scale multi-label data set from the acquired satellite data.

The beginning was made by a module for concurrent atmospheric correction. After reviewing the available algorithms we selected ESA's Sen2Cor processor and modified it in such a way that we could integrate it into Spark. In view of the efforts involved we recommend that for future research the existing alternatives be thoroughly re-examined. Despite an active developer community the Sen2Cor source code is not publicly accessible. Only via detours we managed to extract it from the provided installer. It is only sparsely documented and partially not readable due to patent protection. The libraries dynamically linked against this code section had to be managed manually. Due to the patent protection it was also not possible to optimize the algorithm for distributed processing, although there would be plenty of potential for this. However, by adding a distributed file system interface we have managed to integrate it into the cluster context and successfully demonstrated the distributed atmospheric correction of thousands of Sentinel-2 products. Specifying input and output files as well as other parameters via a central job configuration file made it possible to start various experiments without changing the code or the command to launch the job. Since our changes to the logging process allowed us to collect Spark executor log files we were able to evaluate various debug and performance information about the Sen2Cor business logic and that of the following stages.

The atmospheric correction was followed by a stage for generating large-scale multi-label image archives from atmospherically corrected products. Using our build and submit workflow, it was easy to deploy libraries that allowed access to remote sensing data into the cluster. We have adapted existing techniques for loading Sentinel 2 data for Apache Spark data structures. Patches were extracted using a fixed-size sliding window. By performing the distributed patch generation process on individual sensor bands (in comparison to whole products) we have successfully managed to maintained a low memory footprint. This was underlined by the repeated memory-related failure when loading whole products in our experiments. Contrary to our expectations, the additional costs for aggregating the single-band patches remained moderate. Nevertheless, we see further potential for future work to speed up the patch generation process by minimizing shuffle operation. In our evaluation it became apparent that integrating external data using the PostGIS spatial database extension requires the most time of all tasks. This large share is probably due to the fact that we only had one central PostGIS instance available, which emphasizes a problem with our approach: Although we can dynamically decide what Python libraries are distributed to the worker nodes we have no control over what other software will be run there. Any change to the installed software requires manual work by a third person with the appropriate access rights. A different arrangement of the analysis setup, with one PostGIS instance on every worker node for example, could not be tested here because our administrator was not available in the corresponding period of time. From a researcher's point of view it would be more desirable to decide independently on the software used. A possible approach would be to use operating-system-level virtualization tools such as Docker (Merkel, 2014 [48]). In recent versions of Hadoop YARN experimental support for Docker has been added. Alternatively, a different cluster manager with stable containerization support, like Apache Mesos [6] or Kubernetes (Bernstein, 2014 [11]) could be used.

Another option would be to use geospatial extensions to Apache Spark, like GeoSpark (Yu et al., 2015 [77]) or GeoTrellis (Kini et al., 2014 [41]), instead of an external system like PostGIS. Both options provide spark-native intersection queries suitable for our labeling task. For future work we recommend to compare them with the approach presented here. There is also potential for improvement for the intersection queries themselves. At the moment we don't consider the size of the overlap with the CORINE shapes, which leads to the fact that even for areas with minimal overlap the corresponding label is adopted. While we initially built the query to contain the overlap percentage along with the labels we finally decided against it for performance reasons. It is also quite easy to calculate this information afterwards with the help of the metadata stored in the PostGIS database.

The corresponding adaptation could therefore also be implemented outside of Spark. Overall, storing the metadata of the different sections of the framework in a common database turned out to be a good idea. This enabled us to link generated patches with their output data and to perform analyses without loading the pixel data.

The dataset generated by us is well suited as training data for machine learning. Subsequent analyses, e.g. a multi-label classification or cloud detection task, could be carried out either in Spark itself, on platforms based on the Hadoop ecosystem or on an external system with support for the respective data types, depending on the method used. For deep learning tasks for example, systems such as Horovod [62] (Sergeev et al., 2018), Apache Submarine [7] or TensorFlowOnSpark [75] could be used.

With the last job presented in this thesis we exported the generated data to a GeoTIFF archive. We were able to successfully work around the small file problem. Especially when working with large amounts of data, however, we recommend to use fast storage solutions like NVMe SSDs and file indexing techniques. Unpacking and searching the large volumes of small image files is otherwise very time-consuming.

Bibliography

[1] Accessed: 2019-07-15. URL: https://sentinel.esa.int/documents/247904/1955685/S2A_OPER_GIP_TILPAR_MPC__20151209T095117_V20150622T000000_21000101T000000_B00.kml.

[2] European Space Agency. *Level-2A Algorithm Overview - Figure 3: Scene Classification Values*. Accessed: 2019-11-27. 2019. URL: https://earth.esa.int/web/sentinel/technical-guides/sentinel-2-msi/level-2a/algorithm.

[3] European Space Agency. *SENTINEL-2 User Handbook*. July 2015. URL: https://sentinel.esa.int/documents/247904/685211/Sentinel-2_User_Handbook.

[4] Apache Software Foundation. *Hadoop: v2.1.0*. Accessed: 2019-09-13. Oct. 2019. URL: https://hadoop.apache.org.

[5] *apache/arrow: v0.15.1*. Accessed: 2019-11-28. Oct. 2019. URL: https://github.com/apache/arrow.

[6] *apache/mesos: v1.9.0*. Accessed: 2019-12-05. Sept. 2019. URL: https://github.com/apache/mesos.

[7] *apache/submarine: v0.3.0*. Accessed: 2019-12-05. Nov. 2019. URL: https://github.com/apache/submarine.

[8] Szele Balint. *The Small Files Problem*. Accessed: 2019-11-26. Feb. 2009. URL: https://blog.cloudera.com/the-small-files-problem/.

[9] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011.

[10] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. "Yaml ain't markup language (yaml$^{\text{TM}}$) version 1.1". In: *yaml. org, Tech. Rep* (2005), p. 23.

[11] David Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[12] M Bossard, Jan Feranec, J Otahel, et al. "CORINE land cover technical guide: Addendum 2000". In: (2000).

[13] Tim Bray. "The javascript object notation (json) data interchange format". In: (2014).

[14] Pete Bunting. *Introduction to ARCSI for generating Analysis Ready Data (ARD)*. Accessed: 2019-11-05. URL: https://www.arcsi.remotesensing.info/tutorials/ARCSI_Intro_Tutorial_compress.pdf.

[15] Peter Bunting and Sam Gillingham. "The KEA image file format". In: *Computers & geosciences* 57 (2013), pp. 54–58.

[16] Peter Bunting et al. "The remote sensing and GIS software library (RSGISLib)". In: *Computers & geosciences* 62 (2014), pp. 216–226.

[17] Akshara Preethy Byju et al. "Approximating JPEG 2000 wavelet representation through deep neural networks for remote sensing image scene classification". In: 2019.

[18] Mingmin Chi et al. "Big data for remote sensing: Challenges and opportunities". In: *Proceedings of the IEEE* 104.11 (2016), pp. 2207–2219.

[19] Charilaos Christopoulos, Athanassios Skodras, and Touradj Ebrahimi. "The JPEG2000 still image coding system: an overview". In: *IEEE transactions on consumer electronics* 46.4 (2000), pp. 1103–1127.

[20] Nick Coghlan and Donald Stufft. *PEP 440 – Version Identification and Dependency Specification*. Accessed: 2019-07-19. 2013. URL: `https://www.python.org/dev/peps/pep-0440/`.

[21] Robert Collins. *PEP 508 – Dependency specification for Python Software Packages*. Accessed: 2019-07-20. 2015. URL: `https://www.python.org/dev/peps/pep-0508/`.

[22] *Copernicus Land Service - Pan-European Component: CORINE Land Cover*. Accessed: 2019-07-19. May 2017. URL: `https://land.copernicus.eu/user-corner/publications/clc-flyer/at_download/file`.

[23] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: `https://dask.org`.

[24] Liesbeth De Keukelaere et al. "Atmospheric correction of Landsat-8/OLI and Sentinel-2/MSI data using iCOR algorithm: Validation for coastal and inland waters". In: *European Journal of Remote Sensing* 51.1 (2018), pp. 525–542.

[25] Mathieu Dugré, Valérie Hayot-Sasson, and Tristan Glatard. "A performance comparison of Dask and Apache Spark for data-intensive neuroimaging pipelines". In: *arXiv preprint arXiv:1907.13030* (2019).

[26] NUTS Eurostat. *Nomenclature of territorial units for statistics*. Accessed: 2019-11-26. 1995. URL: `https://ec.europa.eu/eurostat/web/nuts/background`.

[27] Apache Software Foundation. *Apache Solr*. Accessed: 2019-07-19. URL: `https://lucene.apache.org/solr/`.

[28] Jean-loup Gailly and Mark Adler. *zlib: v1.2.11*. Accessed: 2019-11-28. Jan. 2017. URL: `http://www.zlib.net/`.

[29] *Graphic technology — Prepress digital data exchange — Tag image file format for image technology (TIFF/IT)*. Standard. ISO 12639:2004. Geneva, CH: International Organization for Standardization, May 2004.

[30] CS Group. *Sen2Cor Configuration and User Manual*. 2019. URL: `http://step.esa.int/thirdparties/sen2cor/2.8.0/docs/S2-PDGS-MPC-L2A-SUM-V2.8.pdf`.

[31] Olivier Hagolle et al. "A multi-temporal and multi-spectral method to estimate aerosol optical thickness over land, for the atmospheric correction of FormoSat-2, LandSat, VENμS and Sentinel-2 images". In: *Remote Sensing* 7.3 (2015), pp. 2668–2691.

[32] Filip Hanik. *The KISS principle*. Accessed: 2019-11-28. URL: `https://people.apache.org/~fhanik/kiss.html`.

[33] Nicolaus Hanowski. *Sentinel Data Access Annual Report 2018*. Tech. rep. Accessed: 2019-07-19. ESA/ESRIN, 2019. URL: `https://scihub.copernicus.eu/twiki/pub/SciHubWebPortal/AnnualReport2018/COPE-SERCO-RP-19-0389_-_Sentinel_Data_Access_Annual_Report_Y2018_v1.0.pdf`.

[34] Jonathan M Hethey. *GitLab repository management*. Packt Publishing Ltd, 2013.

[35] James N Hughes et al. “Geomesa: a distributed architecture for spatio-temporal fusion”. In: *Geospatial Informatics, Fusion, and Motion Video Analytics V*. Vol. 9473. International Society for Optics and Photonics. 2015, 94730F.

[36] Anaconda Inc. *conda/conda: v4.7.12*. Sept. 2019. URL: `https://github.com/conda/conda`.

[37] Yelp Inc. *Yelp/mrjob: v0.6.12*. Accessed: 2019-09-13. Oct. 2019. URL: `https://github.com/Yelp/mrjob`.

[38] *Information technology – Open Systems Interconnection – Remote Procedure Call (RPC)*. Standard. ISO/IEC 11578:1996. Geneva, CH: International Organization for Standardization, Dec. 1996.

[39] *Information technology — Database languages — SQL multimedia and application packages — Part 3: Spatial*. Standard. ISO/IEC 13249-3:2016. Geneva, CH: International Organization for Standardization, Jan. 2016.

[40] Eran Kampf. *Best Practices Writing Production-Grade PySpark Jobs*. Jan. 2017. URL: `https://developerzen.com/best-practices-writing-production-grade-pyspark-jobs-cb688ac4d20f?gi=ec23eede13de`.

[41] Ameet Kini and Rob Emanuele. “Geotrellis: Adding geospatial capabilities to spark”. In: *Spark Summit* (2014).

[42] *kraftek/awsdownload: c7e60e0*. Accessed: 2019-11-05. 2018. URL: `https://github.com/kraftek/awsdownload`.

[43] Songnian Li et al. “Geospatial big data handling theory and methods: A review and research challenges”. In: *ISPRS journal of Photogrammetry and Remote Sensing* 115 (2016), pp. 119–133.

[44] Jérôme Louis et al. “Sentinel-2 sen2cor: L2a processor for users”. In: *Proceedings of the Living Planet Symposium, Prague, Czech Republic*. 2016, pp. 9–13.

[45] Yan Ma et al. “Remote sensing big data computing: Challenges and opportunities”. In: *Future Generation Computer Systems* 51 (2015), pp. 47–60.

[46] Antonios Makris et al. “Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data.” In: *EDBT/ICDT Workshops*. 2019.

[47] Hector Muro Mauri et al. “Benchmarking Apache Spark spatial libraries”. In: *9th International Congress on Environmental Modelling and Software*. Brigham Young University. 2018.

[48] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[49] Bruce Momjian. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York, 2001.

[50] *NixOS/patchelf: v0.10*. Accessed: 2019-11-05. Mar. 2019. URL: `https://github.com/NixOS/patchelf`.

[51] *olivierhagolle/peps_download: 19c3b3d*. Accessed: 2019-11-05. 2019. URL: `https://github.com/olivierhagolle/peps_download`.

[52] *pdoc3/pdoc: v0.7.1*. Accessed: 2019-11-05. 2019. URL: `https://github.com/pdoc3/pdoc`.

[53] Inc. PKWARE Phil Katz. *ZIP File Format Specification*. Accessed: 2019-11-28. Mar. 1989. URL: `https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT`.

[54] *pyarrow: v0.15.1*. Accessed: 2019-11-05. 2019. URL: `https://pypi.org/project/pyarrow/`.

[55] *pytest-dev/pytest: v5.2.2*. Accessed: 2019-11-05. 2019. URL: `https://github.com/pytest-dev/pytest/`.

[56] Paul Ramsey et al. "Postgis manual". In: *Refractions Research Inc* 17 (2005).

[57] Kenneth Reitz. *Repository Structure and Python*. Accessed: 2019-01-15. 2013. URL: `https://www.kennethreitz.org/essays/repository-structure-and-python`.

[58] Niles Ritter and Mike Ruth. "The GeoTiff data interchange standard for raster geographic images". In: *International Journal of Remote Sensing* 18.7 (1997), pp. 1637–1647.

[59] *sat-utils/sat-api: v0.3.0*. Accessed: 2019-11-05. 2019. URL: `https://github.com/sat-utils/sat-api`.

[60] *sentinelhub-py/sentinelhub: v2.6.0*. 2019. URL: `https://github.com/sentinel-hub/sentinelhub-py`.

[61] *sentinelsat/sentinelsat: v0.13*. Version v0.13. `https://doi.org/10.5281/zenodo.2629555`. Apr. 2019. DOI: `10.5281/zenodo.2629555`.

[62] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[63] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation". In: *Facebook White Paper* 5.8 (2007).

[64] Gencer Sumbul et al. "BigEarthNet: A Large-Scale Benchmark Archive For Remote Sensing Image Understanding". In: *arXiv preprint arXiv:1902.06148* (2019).

[65] Gencer Sümbül and Begüm Demir. "A Novel Multi-Attention Driven System for Multi-Label Remote Sensing Image Classification". In: *IEEE International Conference on Geoscience and Remote Sensing Symposium Yokohama, Japan. Accepted for publication.* 2019.

[66] Microsoft OData Team. *Open Data Protocol*. Accessed: 2019-07-19. URL: `https://www.odata.org/`.

[67] *toml-lang/toml: v0.5.0*. Accessed: 2019-11-15. July 2018. URL: `https://github.com/toml-lang/toml`.

[68] Matthew A Toups. "A study of three paradigms for storing geospatial data: distributed-cloud model, relational database, and indexed flat file". In: (2016). URL: `https://scholarworks.uno.edu/cgi/viewcontent.cgi?article=3292&context=td`.

[69] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[70] Daniele Varrazzo. *psycopg2: v2.8.4*. Accessed: 2019-12-01. Oct. 2019. URL: `http://initd.org/psycopg/`.

[71] Gregory K Wallace. "The JPEG still picture compression standard". In: *IEEE transactions on consumer electronics* 38.1 (1992), pp. xviii–xxxiv.

[72] Frank Warmerdam. "The geospatial data abstraction library". In: *Open source approaches in spatial data handling*. Springer, 2008, pp. 87–104.

[73] *wget: 1.20.3*. Accessed: 2019-07-19. Apr. 2019. URL: `https://www.gnu.org/software/wget/`.

[74] Florian Wilhelm. *Managing isolated Environments with PySpark*. Accessed: 2019-09-13. Mar. 2018. URL: `https://florianwilhelm.info/2018/03/isolated_environments_with_pyspark/`.

[75] *yahoo/TensorFlowOnSpark : v2.0.0*. Accessed: 2019-12-05. Oct. 2019. URL: `https://github.com/yahoo/TensorFlowOnSpark`.

[76] *yaml/pyyaml: v5.1.2*. Accessed: 2019-11-05. July 2019. URL: `https://github.com/yaml/pyyaml`.

[77] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. “Geospark: A cluster computing framework for processing large-scale spatial data”. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2015, p. 70.

[78] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: `10.1145/2934664`. URL: `http://doi.acm.org/10.1145/2934664`.

Appendices

A. Data Acquisition

A.1. Metadata Storage

A.1.1. OpenSearch Response Structure

The following fields were returned by the Copernicus OpenSearch endpoint:

- `beginposition`
- `processinglevel`
- `orbitnumber`
- `producttype`
- `platformname`
- `processingbaseline`
- `size`
- `cloudcoverpercentage`
- `sensoroperationalmode`
- `instrumentshortname`
- `title`
- `orbitdirection`
- `filename`
- `link_alternative`
- `gmlfootprint`
- `platformidentifier`
- `tileid`
- `hv_order_tileid`
- `format`
- `relativeorbitnumber`
- `datatakesensingstart`
- `link`
- `footprint`
- `s2datatakeid`
- `uuid`
- `instrumentname`
- `link_icon`
- `platformserialidentifier`
- `endposition`
- `summary`
- `ingestiondate`
- `identifier`

B. Processing

B.1. Atmospheric Correction

B.1.1. Sen2Cor Dependencies

The following Python dependencies were required in order to run the Sen2Cor v2.8 processor outside of the environment provided by it's standalone installer.

- `cloudpickle` v0.8.1
- `contextlib2` v0.5.5
- `cycler` v0.10.0
- `decorator` v4.4.0
- `funcsigs` v1.0.2
- `GDAL` v2.3.3
- `Glymur` v0.8.17
- `kiwisolver` v1.0.1
- `lxml` v4.3.3
- `matplotlib` v2.2.4
- `mock` v2.0.0
- `networkx` v2.2
- `numexpr` v2.6.9
- `numpy` v1.16.2
- `pbr` v5.1.3
- `Pillow` v5.4.1
- `psutil` v5.6.1
- `pyparsing` v2.3.1
- `python-dateutil` v2.8.0
- `pytz` v2018.9
- `PyWavelets` v1.0.2
- `scikit-image` v0.14.2
- `scipy` v1.2.1
- `six` v1.12.0
- `subprocess32` v3.5.3
- `tables` v3.5.1
- `toolz` v0.9.0
- `dask` v1.2.2

C. Analysis

C.1. Land Cover & Country Label Annotation

C.1.1. CLC 18 Covered Countries

The following 39 countries are covered by the CLC 18 dataset: Albania, Austria, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Cyprus, Czechia, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Kosovo, Latvia, Liechtenstein, Lithuania, Luxembourg, North Macedonia, Malta, Montenegro, Netherlands, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, United Kingdom.

C.1.2. CLC 18 Land Cover Classes

The 44 land cover classes included in the CLC 18 inventory can be divided into five major groups:

1. Artificial surfaces
    - Continuous urban fabric
    - Discontinuous urban fabric
    - Industrial or commercial units
    - Road and rail networks and associated land
    - Port areas
    - Airports
    - Mineral extraction sites
    - Dump sites
    - Construction sites
    - Green urban areas
    - Sport and leisure facilities
    - Unknown
2. Agricultural areas
    - Non-irrigated arable land
    - Permanently irrigated land
    - Rice fields
    - Vineyards
    - Fruit trees and berry plantations
    - Olive groves
    - Pastures
    - Annual crops associated with permanent crops
    - Complex cultivation patterns
    - Land principally occupied by agriculture with significant areas of natural vegetation
3. Forests and semi-natural areas
    - Agro-forestry areas
    - Broad-leaved forest
    - Coniferous forest
    - Mixed forest
    - Natural grassland
    - Transitional woodland/shrub
    - Sclerophyllous vegetation
    - Bare rock
    - Sparsely vegetated areas
    - Burnt areas
    - Glaciers and perpetual snow

4. Wetlands
    - Moors and heathland
    - Beaches and dunes and sands
    - Inland marshes
    - Peatbogs
    - Salt marshes
    - Salines
    - Intertidal flats
5. Waterbodies
    - Water courses
    - Water bodies
    - Coastal lagoons
    - Estuaries
    - Sea and ocean

D. Evaluation

D.1. Data Acquisition

D.1.1. Selection Season Statistics

Scale	**Spring**	**Summer**	**Autumn**	**Winter**
small	33.33%	33.33%	0.0%	33.33%
medium	35.80%	19.89%	25.00%	19.32%
large	25.85%	27.41%	24.26%	22.49%

Table D.1.
Evaluation data selection season statistics