Utilizing the Hadoop Ecosystem to Improve In-house Data Analysis of Spacecraft Housekeeping Data

Author
Esther Kok
s4140389
me@estherkok.com

Radboud Supervisor
Prof. Arjen de Vries
arjen@cs.ru.nl

ESA Supervisor
Dr. Rui Santos
rui.santos@esa.int

January 16, 2020
Acknowledgements

I’d like to take this opportunity to thank James and Rui for giving me this great project to work on at ESOC. Not only was it an interesting and challenging project, it also gave me the opportunity to work for ESA (which has been a long standing dream of mine), and allowed for my continued involvement in ESA projects as a contractor.

Thank you, Arjen, for being my supervisor for this thesis project. Not only did you give me valuable input, you also gave me the time and space to complete it next to all the other things that were going on in my life.

To my wonderful coworkers at ESOC, I’d also like to say thanks. You made my time at ESOC memorable. The endless discussions, lunches, board game sessions, after work flights and everything in between will always be among my fondest memories.

Stefan, I know that I’ve said it on a number of occasions, but thank you. Your support has made it possible for me to go on this endeavor while simultaneously knowing that everything at home would still be standing in my absence. You are my rock and my hero.
Abstract

At the European Space Operations Centre (ESOC) in Darmstadt, Germany, all activity focuses on controlling and managing spacecrafts and their ground support systems. Part of the activities at ESOC include the implementation of new technologies to deal with an increase in data rate coming from newer spacecrafts.

The Data Analytics Team for Operations (DATO) group uses housekeeping data from the spacecraft to look for possible causes of anomalies and to assist spacecraft operators in diagnosing problems. However, access to data stored in the different data lakes at ESOC is not very intuitive for the machine learning engineers, as well as access to the hardware to speed up processing capabilities.

The work in this thesis addresses the bottlenecks in access to the data lakes and describes the implementation of a prototype Python library to act as a solution for these bottlenecks. The prototype not only enables data retrieval from data lakes to a local machine, but also allows for the submission of Spark data processing jobs. This helps speed up the process of data analysis for the machine learning engineers by using the available Hadoop infrastructure.

Performance of the prototype is measured with experiments based on two use cases; the Mars Express and Aeolus. Though the experiments show that there are a number of limitations with the implementation of the prototype, mostly due to constraints on the software baseline, they also show its potential for production grade data analytics job submissions to the Hadoop architecture.
# Contents

List of Tables VI

List of Figures VII

Glossary IX

1 Introduction 1

1.1 ESOC ................................. 1

1.2 Problem Statement ................................. 1

1.3 Research Question ................................. 2

1.4 Performed Work ................................. 3

1.5 Thesis Overview ................................. 3

2 ARES Data Layout 4

2.1 Operational Purpose ................................. 4

2.2 Database Infrastructure ................................. 6

2.2.1 HBase ................................. 6

2.2.2 Google Protocol Buffer ................................. 7

2.3 Data Types and Storage ................................. 8

2.3.1 Telemetry Parameters ................................. 8

2.3.2 Other Data Types ................................. 9
3 Data Availability

3.1 Chapter Goal ................................................. 10
3.2 Moving Calculations, not Data ................................ 10
3.3 Data Access at ESOC ..................................... 12
3.4 Bottlenecks at ESOC ..................................... 13
  3.4.1 Discussion Setup .................................. 13
  3.4.2 Discussion Outcome ................................ 13
  3.4.3 Solution Proposals ................................ 14

4 PyAres ......................................................... 16

4.1 Library Goal ............................................... 16
4.2 Core Components ......................................... 17
  4.2.1 Data Retriever ..................................... 18
  4.2.2 Job Submitter ..................................... 19
  4.2.3 Result Retriever .................................. 24
4.3 Operational Deployment ................................ 25
4.4 Limitations ................................................ 25
  4.4.1 Software Baselines ................................ 25
  4.4.2 Result Persistence ................................ 26
5 Use Cases

5.1 Use Case Goals ................................................................. 27
5.2 Mars Express ................................................................. 28
  5.2.1 Mission ................................................................. 28
  5.2.2 MEX Challenge ........................................................ 29
  5.2.3 PyAres Implementation ............................................. 31
5.3 Aeolus ................................................................. 34
  5.3.1 Mission ................................................................. 34
  5.3.2 Spacecraft Status Report ......................................... 35
  5.3.3 PyAres Implementation Performance .......................... 35

6 Discussion ........................................................................ 38

6.1 Research Questions ....................................................... 38
6.2 Bottlenecks ............................................................... 39
6.3 PyAres Limitations ....................................................... 39
  6.3.1 Happybase, Thrift and Protobuf ......................... 39
  6.3.2 HDFS ................................................................. 40
  6.3.3 Spark ................................................................. 40
  6.3.4 Generalization ....................................................... 41
6.4 Future Work ............................................................. 42
6.5 Conclusions ............................................................. 42
6.6 Contributions .......................................................... 43

References ........................................................................ 44

Appendices ......................................................................... 47

A PyAres API .................................................................... 48
List of Tables

3.1 Question guide for the semi-structured interviews conducted to explore the bottlenecks the engineers experienced in their interactions with the housekeeping data from the ARES data lakes. ................................................................. 15

5.1 Specifications of the machines for the client and the cluster. The cluster has a number of nodes with the same specifications, where the client is a single virtual machine. . . . . 31

5.2 For n parameters with a a certain time period, the job resulted in an above described number of processed samples. ................................................................. 37
List of Figures

2.1 ARES environment and architecture (Source: ARES Software Design Document (ARES team, 2017a)) ................................................................. 5

2.2 HBase Architecture. HMaster communicates with client and resource managers to manage the data and the region servers. ........................................ 7

4.1 PyAres Architecture of the core components. The data retriever accesses the ARES database, enabling data retrieval to a local machine, or to retrieve data to the Spark job. The job submitter defines and submits Spark jobs to the YARN scheduler to run on the cluster. Job results get persisted in HDFS and can be accessed on a local machine via the result retriever. ................................................................. 17

4.2 PyAres Architecture of the data retriever component. The parameter_sample_provider.py accesses the ARES MariaDB as the source of the parameter ID via data_source.py, then translates the query to a bytebuffer via byte_buffer.py. This bytebuffer is the row key to define the query for HBase, which is accessed via the hbase_connect.py functionality. The database is scanned using the query and returns data in Protobuf form, which is decoded into parameter sample objects via the protobuf.py and sample.py functionality. The data can be returned to either a local machine or further used as part of a data processing job with Spark. ................................................................. 18

4.3 PyAres architecture of the job submitter component. The ares_job.py functionality takes care defining a Spark job suitable for processing ARES data, and helps the user submit it to the cluster. It offers functionality to define a job that calculates user defined functions or a predetermined set of statistics for each parameter. The job_helper.py takes care of retrieving data via the parameter_sample_provider.py, and also contains functionality to help with the processing of the data inside the job executing on the Spark nodes. .... 20

4.4 PySpark Data Flow architecture. The locally initialized Spark Context translates Python data types through a Py4J socket to JVM compatible data types, which is launched on initialization of the Spark Context. On the cluster, the JVM types are transformed to Python through a pipe that sends a pickle with user code and data to be processed to the Python sub-processes. Source: (Apache Spark Team, 2016) .... 21
4.5 PySpark submit command code. This command is composed and executed as part of the job execution functionality.

4.6 PyAres Architecture of the result retrieval component. Results are stored in Parquet and returned to the user as a pandas dataframe.

5.1 MEX orbit around Mars with schematic representation of different shadows cast by the sun on the spacecraft. Marked are the occurrences of the different events in the data relative to the location of the spacecraft in the shadow. (ESA, 2016)

5.2 Illustration of the MEX orbiter and the axes x, y and z, which correspond to front, left and up. These axes can be used to derive back, right and down. \(\alpha_x\) is the solar aspect angle of the front side, i.e., the angle between the normal \(\tilde{\nu}_x\) and the Sun-MEX line. \(\tilde{\nu}_p\) marks the normal of the panels (Petkovic, M., Boumghar, R., Breskvar, M., Dzeroski, S., Kocev, D., Levatic, J., Lucas, L., Osojnik, A., Zenko, B., Simidjievski, N., 2018).

5.3 Performance of the data transportation to the prototype cluster (top) and a client machine (bottom), measured in seconds. The size of the parameter Scan Buffer Size (500, 1000, 2500, 5000, 10000, 50000, 100000, 500000), e.g. the number of samples that can be retrieved per scan, is plotted against the Connection Pool Size (1, 10, 25, 50, 100), e.g. the number of connections to HBase the application has available for retrieving data.

5.4 Scalability of processing time of MEX use case. The two series 'client' and 'cluster', point to the different implementations of the use case, where the 'client' series signifies the data transportation to a client machine after which the data is processed, and the 'cluster' series points to the Spark job submitted to the prototype cluster. Data is processed to a resolution of one hour for five time periods (one day, one week, one month, one year and four years).

5.5 Laser power telemeter of the Aeolus payload Aladin in mJ, plotted with five minute resolution for one week from Monday 3 December 2018 to Saturday 8 December 2018.

5.6 Performance of the data retrieval to the cluster, measured in seconds.
Glossary

API Application Programming Interface. 4
ARES Analysis and Reporting System. 2
ASAF ARES Spark Analysis Framework. 2
CLI Command Line Interface. 20
DARC Data Archive. 4
DATO Data Analysis Team for Operations. 2
DrMUST MUST client for anomaly detection. 2
EBI European Bioinformatics Institute. 11
EDDS EGOS Data Dissemination System. 4
EGOS ESA Ground Operation System. IX
ESA European Space Agency. 1
ESAC European Space Astronomy Centre. 12
ESOC European Space Operations Centre. 1
ESRIN European Space Research INstitute. 12
FARC File Archive. 4
MEX Mars Express. 28
MMI Man-Machine Interface. 4
MUST Mission Utility and Support Tools. 2
NCBI National Center for Biotechnology Information. 11
OPS LAN  The network environment with limited or w/o any incoming connectivity from outside for operational security. 11

OPS-GIM  Operations Ground Infrastructure Management. 2

PARC  Packet Archive. 4

SLES  Suse Linux Enterprise Server. 22

SpaCon  Spacecraft Controller. 35

SRS  Software Requirement Specification. 22

TM  Telemetry. 8

UDF  User Defined Function. 21

WebMUST  Web interface to MUST. 2
Chapter 1

Introduction

1.1 ESOC

At the European Space Operations Centre (ESOC) in Darmstadt, Germany, all activity focuses on controlling spacecrafts in orbit, managing the global tracking station network, and designing and building systems on the ground that support missions in space. ESOC is the day to day operational center of ESA (ESA, 2018a).

ESOC is home to a small number of groups that focus on infrastructure, flight dynamics, space debris, communication with spacecrafts etc. OPS-GIM is the group responsible for mission transcending infrastructure and providing infrastructure services for each mission. Part of their responsibilities is to implement new technologies for changing or future requirements, like big data solutions for handling an increase in data rate of the newer missions.

1.2 Problem Statement

At ESOC housekeeping data from the countless missions is stored to analyze the health of the spacecraft and to backtrack possible causes of anomalies. Housekeeping data comes in the form of data samples, which can range from telecommands to telemetry. Each sample consists of a timestamp, a sample name if applicable, e.g. telemetry samples have a corresponding parameter name, whereas telecommands do not, and the value of the sample, which can be a string or an integer.

For a single mission with a lifetime of 10 years and a nominal parameter sample rate of 1000 samples/second, the mission will have to store approximately a total of 315360M samples. With the newer missions this number is only increasing (Santos, 2014), due to improvements of on-board
technology, the increase of space link rates and mission complexity. For example; GAIA (ESA, 2018d) has a sample rate of approximately 1700 samples/second.

The Analysis and Reporting System (ARES) has a fairly recently implemented Hadoop HBase instance for the storage of sample data (Santos, 2014), but running data analyses or machine learning tasks with this data is not part of the ARES system and are still being run locally/non-distributed. Data scientists working with the data stored in ARES don’t access the data via an interface to HBase, but pull the data into their own MySQL database via the web-interface to the Mission Utility and Support Tools (MUST, WebMUST).

The prototype ARES Spark Analysis Framework (ASAF), currently running on a development cluster, aims to transform the data analytics process to a distributed one. However, the implementation covers only a single specific tool (DrMUST), which is used for outlier detection, and still leaves a lack of interface to the data from outside a single application or tool.

For new missions like BepiColumbo (ESA, 2018c) and Aeolus (ESA, 2018e) the Ground Infrastructure Management group (OPS-GIM) would like to move the ASAF prototype to a more mature implementation, where parameter data storage and analysis is centralized and generalized for the missions, instead of having customized or ad hoc data storage and analysis for each of them.

1.3 Research Question

The machine learning group would like to be able to use their custom Python code on the ARES framework, as well as approach the ARES database directly. Compatibility with Python code would be a great asset to them, because of their use of Pandas dataframe for analysis. The main motivation for this thesis project is to identify the reasons for the machine learning engineers from the Data Analytics Team for Operations (DATO) group not using the local big data lakes, and to improve on that situation. In this thesis I will identify the blocking points and trade-offs in access to the big data lakes and infrastructure to match ESOC operational data analysis requirements. Specific sub-questions include:

- How are the machine learning engineers at ESOC using the big data lakes?
- What are the bottlenecks in data access for the machine learning engineers?
- What are the bottlenecks in result persistence?
- How well does the current data layout generalize for algorithms beyond MUST and can it be improved upon?
- What is the cost of (not) adapting client code to the big data platform, e.g. machine learning code in Python?
- How can we make the ARES interface usable for the DATO group at ESOC?
- Can we improve on data pre-processing using the ARES interface?
1.4 Performed Work

While the research in this thesis aims to analyze and solve problems with usage of local data lakes, its use case is limited to the data layout and availability at ESOC. The advantage being that the solution can be found in a practical implementation that transcends theoretical platitudes, while at the same time the disadvantage remains that it might be hard to generalize for other use cases.

In the end the work performed in the scope of this thesis aims at making improvements on the data availability and processing currently offered by ARES compatible interfaces, by creating a Python interface for the DATO group. The work done at ESOC OPS-GIM formed the practical background for the research described in this thesis. Practical work included:

- Refactoring the ARES data provisioning API to Python.
- Combining data provisioning and distributed computing functionality into one Python library: PyAres
- Persisting and retrieving results from PyAres jobs to HDFS and back.
- Deploying the new PyAres library in a Docker image.

The code resides on the local ESOC code repository, but a fork also resides on a private repository on GitHub. Access can be granted on request to specific persons.

1.5 Thesis Overview

This thesis coincides with the above mentioned work performed for ESOC OPS-GIM. Chapter 2 provides theoretical background and implementation details on the current state of ARES and it’s components. Chapter 3 describes blocking points in accessing data lakes in general, as well as ESOC specific issues and bottlenecks. The subsequent chapters (4 and 5) describe the prototype solution and the results from the adapted use cases. The last chapter, chapter 6, discusses the shortcomings of this research and concludes the results of the work.
Chapter 2

ARES Data Layout

2.1 Operational Purpose

ARES is the Analysis and Reporting System. It is part of the ground segment architecture and the off-line environment for the mission control system infrastructure at ESOC. Off-line in this context signifies that the incoming data from the spacecraft is not directly passed through the system, but the system receives data from secondary sources. As ARES is part of the ground segment, there are redundancies in place and development moves through the prototype and test cluster before moving to the production cluster (Santos, Marques, & Eggleston, 2017).

ARES facilitates operational data analysis to support the troubleshooting of operational events, the performance of space and ground systems, and the generation of specific reports. Its main components take care of the storage, retrieval and manipulation of the operational data in a persistent way, as well as providing a man-machine interface (MMI) that can display and analyze the operational data (ARES team, 2017a).

The central point of the ARES system is the ARES database. As seen in figure 2.1, the database is populated with operational data coming from different data sources (PARC, FARC, DARC) through the EGOS Data Dissemination System (EDDS) in scheduled operations, or from local files via manual import operations, using the ARES importer. The ARES MMI and other ARES clients access the operational data through the ARES data provision API, in order to retrieve and analyze it further (ARES team, 2017a).

The primary purpose of the EDDS is to provide controlled access to science and non-science mission control system data to users who do not have access to the mission control systems themselves. The ARES importer uses the Spark component of the Hadoop ecosystem ARES is built on to schedule jobs that import the data from EDDS into the ARES database.
Figure 2.1: ARES environment and architecture (Source: ARES Software Design Document (ARES team, 2017a))
The ARES database is actually a dual database system with an instance of MariaDB and an HBase cluster. Each mission has their own instance of the ARES database deployed. The mission metadata and ARES service and management data are kept in MariaDB and the mission sample data is kept in the scalable HBase database cluster. The ARES database contains tables for the operational data, like telemetry and parameters, and their samples, as well as telecommands and others. The ARES data importer assigns primary keys to the imported data and stores these in the MariaDB tables, before using this primary key to form part of the HBase rowkey. The data stored in HBase is wrapped in a data type specific combination of byte buffer and Google Protocol Buffer.

2.2 Database Infrastructure

As mentioned, the ARES database uses a combination of MariaDB, HBase and Google Protocol Buffer to store the data. This section describes some of the background of these technologies. The data type specific layout will be discussed in section 2.3.

2.2.1 HBase

Googles Bigtable, and its derivate HBase, use a data model that is a "sparse, distributed, persistent multidimensional sorted map" (Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E., 2006). This definition can be separated to understand the general idea behind the HBase model.

Map An HBase table is in its core a map or an associating array. It is an abstract data type composed of a collection of keys and a collection of values, where each key is associated with one value, similar to a Python dictionary (Apache HBase team, 2018; Wilson, 2008).

Sparse The nature of the table, where columns are optional, means that a great number of rows will have no data for certain columns, making it a sparse structure (Apache HBase team, 2018; Wilson, 2008).

Distributed An HBase table is distributed, it is spread out among an array of independent machines, on top of HDFS, with a number of replicates. It is stored in a sorted manner based on the row key (Apache HBase team, 2018; Wilson, 2008).

Persistent The data is stored on disk and tends not to get deleted or modified (Apache HBase team, 2018; Wilson, 2008).
The maps can be nested because of their column family and qualifier. The families are defined at the creation of the table and virtually immutable, but the qualifiers are not. The dimension time is also included in each cell, attached to the value of the cell. A basic row could be described as \{rowkey : \{column\_family : \{column\_qualifier : \{time : value\}\}\}\} (Apache HBase team, 2018; Wilson, 2008).

The HBase architecture, as seen in figure 2.2, shows how this definition comes together in the architecture itself. The data is distributed over different region servers in their sparse, multidimensional maps. The data is persisted over different region servers to ensure redundancy in storage. The HMaster keeps track of the metadata, what data is stored where, and the access of the data itself to clients and resource managers.

2.2.2 Google Protocol Buffer

The data stored in HBase is wrapped in a Google protocol buffer. Google’s protocol buffers are language-neutral, platform-neutral, extensible mechanisms for serializing structured data (Varda, 2008). The protocol buffer encodes the data in a compact way that is easy to transform back to its original form. It makes use of a schema to create a so called message, which is a small logical record of key value pairs to store the information. The data schema can be used between languages as well, making it easy to pack and unpack data between for example Java and Python objects. This language independence and compactness makes protocol buffers very suitable for long term storage, where the language interacting with the data might change or evolve over time.
2.3 Data Types and Storage

The domain of Space Operations at ESA produces four different data types that are processed by ARES and stored in the ARES database. Each data type requires their own data layout in the ARES database. As mentioned before, the ARES database is a combination of MariaDB, HBase and Google Protocol Buffer technologies. Though the types of the data differ, they are all time-series based.

2.3.1 Telemetry Parameters

As described by Wilfrid J. Mayo-Wells in *The Origin of Space Telemetry*, "[Telemetry] is the art of measurement at a distance in which the physical quantity being measured is converted to another quantity for transmission to receiving and recording equipments (Mayo-Wells, 1963)."

In the context of aerospace technology, a telemeter is a device providing a single point of measurement in the spacecraft, used to measure the operational status of a certain piece of equipment. The telemeter can be responsible for measuring a current, or the temperature of a part, or record the status (ON/OFF), etc. The (digital) measurement is transformed inside the spacecraft into a signal in the electromagnetic spectrum, e.g. radio waves, after which it is received by the ground station and transformed again into a digital signal.

The received data packets, once decoded, can contain telemetry parameter data, also known as TM parameter data. In spacecraft operations TM data consists of all the measured data coming from the spacecraft that is not involved in the science mission. TM parameter data is the single most important type of data to keep track of spacecraft health and to assess spacecraft performance. A TM parameter sample is a single measurement at a given time. These samples can consist of the raw values coming from the spacecraft, but usually they are processed before they are stored, at which point there are called engineered values. One of the common processing steps is to calibrate the measurement. When speaking about telemetry samples, spacecraft engineers usually refer to the engineered values, not the raw values. A TM parameter sample in stored in ARES as an HBase record usually contains both the raw and the engineered value (ARES team, 2017a).

Aside from regular TM parameters, there are also synthetic TM parameters. These are the parameters for which further processing steps have been undertaken, like a specific function or formula that gets passed over the value. Information about the origin of the sample, e.g. raw, synthetic or otherwise, is stored in the metadata database (ARES team, 2017a).

MariaDB For the ARES data provision API to correctly retrieve parameter samples, there are two MariaDB tables that need to be accessed: DATA.DEFS.TBL and DATA.DEFS.TYPE.TBL.

DATA.DEFS.TBL contains the TM parameter definitions. This table corresponds the parameter ID primary key to the name and description of the parameter. Every record in the table has attributes PID, NAME, DESCRIPTION, ENGVALUNIT, DATACATEGORY, RAW_DATACATEGORY, ACTIVE and SYSTEM_ELEMENT.

DATA.DEFS.TYPE.TBL is the table that defines the data types and relates DATA.DEFS.TBL DataCategory and RawDataCategory identifiers to the type of a data definition.
HBase  As described in section 2.2.1, HBase stores its data in an abstract data type that consist of a collection of keys and values. The identifier of the collection is called the row key. For ARES’ HBase telemetry sample table **ParamSamples**, the row key is a byte buffer containing a reference to the parameter id in the MariaDB table (first 3 bytes), plus the timestamp of the parameter in Unix Epoch microseconds (8 bytes). The order in which the row key is assembled means that the data is sorted on parameter first and then on time. This design choice was made to optimize the reading speed from the table, where the data is read sequentially, because queries to the database are also based on this pattern. A normal query would be to retrieve all samples for parameter A in the time period X to Y. The value associated with the row key is the sample itself in column ”v:e”. A basic row in ParamSamples could look like this: `{row_key: {v: {e: {HBase_timestamp: sample_value}}}}`.

### 2.3.2 Other Data Types

For completeness the other data types are described here. Since they were not part of the prototype implementation they will not be described in detail.

**Telecommands**  A telecommand is uploaded to the spacecraft to make the spacecraft execute one or a sequence of activities, for example a maneuver that lowers the orbit of the spacecraft. The executed telecommands are stored in ARES with their Unix Epoch timestamp associating to upload time.

**Events**  Events are outputs triggered by (on board) software to give information about the status of the spacecraft or internal software. These events are typically stored in event logs, but are also stored in ARES. The ARES records contain event id and event severity for event related to the spacecraft, mission control system or mission planning system, as well as the Unix Epoch timestamp of internal event occurrence.

**Telemetry and Telecommand Packets**  Packets are encoded byte sequences that contain metadata and either the telecommands, which is the case for packets send towards the spacecraft, or telemetry, which is the case for packets received from the spacecraft. The packets are the original raw data source, which is persisted in ARES.
Chapter 3

Data Availability

3.1 Chapter Goal

This chapter addresses the research questions

1. How are the machine learning engineers at ESOC using the big data lakes?
2. What are the bottlenecks in data access for the machine learning engineers?
3. What are the bottlenecks in result persistence?

Data availability is a challenge wherever data is used, and at ESOC specific challenges exist in both data access and data persistence. The following sections will address general issues in data availability and access, and look at the specific issues at ESOC in terms of bottlenecks in both access to data and persistence of results. Part of the analysis of the issues at ESOC was performed using a semi-structured interview.

3.2 Moving Calculations, not Data

Researchers in the empirical sciences like physics and biology usually gather their data on a local machine, perform their analysis to create the results and persist the results either locally or in some sort of central database. This was manageable when calculations only required a few MBs of data, but as the tools to gather data became more sophisticated and cheaper, so did the amount of data increase.
With the increasing amount of data being gathered, stored, and processed, the nature of data processing also changed. Initiatives like the Sloan Digital Archive (Sloan Digital Sky Survey, 2018), NCBI (National Center for Biotechnology Information, 2019), and EBI (European Bioinformatics Institute, 2019) have improved access to these digital archives, and offer access to calculations on the data side as well. However, because it is hard to break habits (Danner, Aarts, & de Vries, 2008), researchers that operate in these empirical sciences are slow in their uptake of this new workflow, and the adoption of the methods are therefore a lot less widespread.

To illustrate, most NCBI workflows consist of using the API to access the data, pull a copy to a local server and do the calculations there (Gray & Szalay, 2004). Sometimes even to the extent of creating a local copy of an entire database, consisting of a few dozen GB. For example, proteomics research using the model organism C. elegans might involve pulling a local copy of the entire C. elegans proteome for the calculations.

At ESA, the largest initiative to provide users with access to data is the Copernicus European Union’s Earth Observation Programme (European Commission, 2019). The initiative was setup by the European Commission as a means to boost industry development and citizen involvement of earth observation applications. It offers access to data from a constellation of Earth observation satellites.

There is still a limitation to these approaches, because the data is still being moved across the internet, which is an expensive operation. In their paper about their work on data management for the Sloan Digital Archive Gray and Szalay et al. (Gray & Szalay, 2004) wrote:

... scientists must learn to move the questions-and-answers across the internet, and do the data analysis near the data whenever possible.

Jim Gray wrote about distributing data over multiple machines in 1992 (Gray & DeWitt, 1992), but only now, with the current data avalanche, has this become a necessity and not a mere theoretically nice idea. By investing in clusters of relatively cheap machines that can perform loads of parallel calculations, instead of one really expensive mainframe, these days it has become much cheaper to move the calculations to the data and not the other way around.

The necessity of moving the calculations to big data architectures has given rise to a new problem: access to these big data infrastructures. At ESOC, while not publicly accessible, the ARES cluster offers a data lake with, among others, the telemetry data samples, and the possibility of performing calculations at the location of the data. ARES’ main use case, to offer a data lake outside of OPS LAN for off-line (post-processing) analysis, was kept clear in mind in the design of the data lake itself (ARES team, 2017a).

However, beyond standardized analyses data access becomes complicated and it is apparent from the design that data science was not one of the end use cases taken into account at the time the framework was designed. One of the common pitfalls of designing any big data architecture is that it is designed without or insufficient eye on the end users (Kiran Prakash & Lucy Chambers, 2019). At ESOC, this causes one the main potential advantages of the ARES infrastructure, performing...
custom calculations at the data side, to be under-utilized. The only fully fledged product that uses the computation functionality is part of the data ingestion into the ARES database, the aforementioned off-line post-processing. There are data view applications provided, but as it is, the direct access to the ARES data is difficult from the (data science) client machine side, and moving the calculations to the data is not supported in a straightforward way.

### 3.3 Data Access at ESOC

As Jim Gray points out, historically speaking astronomy and related sciences are very empirical in nature (Gray & Szalay, 2004). Scientist gathered and processed their own data. This is also the case for space operations. Even though computations have sped up processes like spacecraft simulations, and cause more and more operational data to be available for analysis, even ahead of the launch, a lot of the new developments in computer science remain untouched. Machine Learning being one of them.

Machine Learning is a fairly new process at ESOC. The main focus lies with space operations and therefor telemetry and other spacecraft housekeeping data, usually coming from the spacecraft platform. This housekeeping data is traditionally in the hands of the spacecraft operations engineers. Housekeeping data has only recently moved towards Big Data infrastructures, in contrast to the science data.

Other ESA sites, like ESAC and ESRIN, have more experience with the Machine Learning discipline because the main focus at those sites lies with the science data, where Machine Learning has played a positive role in understanding the increasing amount of available science data coming from the spacecraft payload. With the development of better cameras, sensors, and data uplinks, the science data has had an explosion in volume. This warranted the rise of dedicated infrastructures to deal with these volumes, and the deployment of data analysis structures on that infrastructures. Initiatives like the Copernicus Program (European Commission, 2019) increased the need to move to a more efficient infrastructure for data storage, access, and processing.

ESOC staying a bit behind in this respect can be attributed in part to the historically lower volume of the housekeeping data, which has in the past not warranted a move to a larger, more efficient infrastructure, but also because the users, e.g. the spacecraft operators and controllers, have had traditionally speaking no need for machine learning over large amounts of data in their day to day operations. Therefore, the data access to smaller database instances, e.g. local SQL databases, was never a problem.

The traditional data access system at ESOC was MUST. MUST pulled the data from OPS LAN into a local MySQL database. With the introduction of ARES, the MUST data imports have adapted to be compatible with the newer missions that are supported by ARES. In these cases MUST is configured to pull the ARES data, stored in HBase, into a local MySQL database, where the MUST MMI can be accessed via the web-interface. Aside from this redundant data storage, the increased focus on machine learning for spacecraft operations is causing the MUST workflow to become problematic.
3.4 Bottlenecks at ESOC

As part of the work described in this document, there were discussions planned with user representatives, both machine learning experts and spacecraft operators. These user representative were chosen based on their involvement with ARES mission data from different perspectives; machine learning experts try to solve a specific problem using the data, where the spacecraft operators try to identify whether problems have arisen in their spacecraft.

The user representatives needed to interact with ARES data, so the spacecraft operators from older missions were not selected, because these missions do not use ARES for their data storage. In the end, operators from Aeolus were selected, due to the fact they were using Python to create their reports and were very interested in learning more about machine learning to aid in their analysis. The machine learning experts all interacted with multiple different missions, so ARES usage was not needed as a selection criteria, and since there are only a small number of dedicated machine learning engineers, they were all selected.

3.4.1 Discussion Setup

The goal of the discussions was to discover the bottlenecks the users experience in their access to ARES data lakes, processing of this data, and bottlenecks in persistence of their analysis results. The discussions were shaped as a semi-structured interview, because of the small set of participants, with the guide described in table 3.1 to help focus the conversation.

3.4.2 Discussion Outcome

One of the topics of the discussion, the usage of existing tools (the ARES and MUST MMI), was discussed elaborately. It became clear that Python is the preferred programming language to interact with the data, because it is an established data science language and for that reason a go-to tool, but neither ARES nor MUST MMI offers support for Python interaction. There are a number of grievances with the ARES and MUST interactions that could be identified:

- ARES and MUST cannot be interfaced with Python, making machine learning and other data analytics related activities difficult without some data import hacks.
- The current machine learning workflow involves data retrieval to local machines via an export of MUST data to a local database or files.
- MUST cannot handle the large volumes of data from modern missions, making it difficult to perform data analysis on either a long time period of data, a large amount of telemeters (a spacecraft can contain up to 40,000 telemeters), or both.
• MUST cannot analyse the large volumes of data, because it is a visualizer, not an analysis tool. The analyses themselves need to be performed outside of the tool, using the MySQL database.

• MUST cannot handle the persistence of the results of large analyses, where the data needs to be stored in local files, because the attached SQL database does not handle the result storage. It is neither flexible enough to handle custom data, nor capable of handling the larger volumes. So users need to create their own local database for result persistence.

3.4.3 Solution Proposals

A part of the discussion was focused on finding solutions for these issues. Since MUST support is no longer available and desirable for new missions, and older missions will move their data to ARES in the future, the decision was made to only describe solutions that involve ARES data. One of the grievances with MUST was the incapability to process large volumes of data, and since the ARES infrastructure offers components capable of processing larger volumes of data, this was discussed as one of the proposed parts of the solution. The following user interfaces for a solution were identified:

• Access to ARES data on local machines for Python based data analytics prototyping.

• A generalized approach for processing large amounts of data on the ARES architecture, as part of the data pre-processing step for the machine learning workflow. From a user perspective this solution should be as least complex as possible, since Spark applications can require very specific knowledge.

• Result storage and availability for reuse and other users, so data should be stored in a centralized way that is accessible for the client.

In summary, the proposed solution can be defined as an application or API that uses Python to give the user access to the ARES data and the results from their data processing, where the result data comes from custom data pre-processing using the Spark component of the ARES infrastructure.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation to Housekeeping Data</td>
<td>What is your function and how does it relate to data analytics?</td>
</tr>
<tr>
<td></td>
<td>What does your workflow and/or day-to-day interaction with housekeeping data look like?</td>
</tr>
<tr>
<td>Usage of Existing Tools</td>
<td>Are involved in a mission that uses the ARES MMI or are you using the MUST MMI?</td>
</tr>
<tr>
<td></td>
<td>If so, what are your experiences?</td>
</tr>
<tr>
<td></td>
<td>If neither, what alternatives are you using and what are your experiences with those?</td>
</tr>
<tr>
<td>Data Interactions</td>
<td>What is the volume of data that you usually interact with in terms of resolution, time period, and number of telemeters?</td>
</tr>
<tr>
<td></td>
<td>What is your usual process of gaining access to the data?</td>
</tr>
<tr>
<td></td>
<td>Are you happy with this process? Why (not)?</td>
</tr>
<tr>
<td></td>
<td>How long does it usually take for you to get the data you need?</td>
</tr>
<tr>
<td>Data Processing</td>
<td>What kind of data processing do you perform in your daily work?</td>
</tr>
<tr>
<td></td>
<td>Is there a specific goal you want to reach with this processing?</td>
</tr>
<tr>
<td></td>
<td>How are you persisting your results?</td>
</tr>
<tr>
<td></td>
<td>Do others need access to your results?</td>
</tr>
<tr>
<td></td>
<td>How long does a task usually take you?</td>
</tr>
<tr>
<td>Potential Solutions</td>
<td>How would you prefer to interact with the data?</td>
</tr>
<tr>
<td></td>
<td>How would you prefer to work on your data analysis?</td>
</tr>
<tr>
<td></td>
<td>Which use cases do you have to test a potential solution?</td>
</tr>
</tbody>
</table>

Table 3.1: Question guide for the semi-structured interviews conducted to explore the bottlenecks the engineers experienced in their interactions with the housekeeping data from the ARES data lakes.
Chapter 4

PyAres

4.1 Library Goal

Based on the outcome of the semi-structured interview mentioned in section 3.4, a solution was proposed, designed and implemented that uses Python to give access to the ARES data on their local machine, allows for Spark based data processing on the ARES cluster, and facilitates the access to the results from the calculations back to the local machine into native Python data structures. PyAres is the result of the implementation effort, and main work performed for this thesis, for the proposed solutions described in section 3.4. It aims to resolve the mentioned data processing blockages in the DATO group. This chapter describes the performed work and will address the research questions

*How can we make the ARES interface usable for the Machine Learning engineers at ESOC?*

and

*Can we improve on data pre-processing using the ARES interface?*

Ultimately the goal of PyAres is to take away the complexity of the ARES framework for the end user, allowing for easy access to ARES data and adding functionality to run custom jobs on the cluster to improve data pre-processing. This chapter describes implementation details for these components to interface with the ARES framework. All of the components described have an API that can be consulted in appendix A.
4.2 Core Components

Previous chapters have described the ARES infrastructure, the stored data layout in the ARES database, as well as mentions of ARES being built on top of the Hadoop ecosystem. The core components of PyAres allow interfacing with the ARES database instances, offer a generic API for the design of custom jobs that can run on the ARES Spark instances, and enables access to HDFS persisted results from said jobs. Figure 4.1 shows the basic architecture of the library and the different ARES components that it interacts with. The following paragraphs describe each of these components in more detail.

A precondition of the deployment of the PyAres library is that the machine needs to be Linux based, running Linux and has to have been configured as an ARES gateway node. Meaning that the machine needs to be able to locate the needed configurations to use the ARES components. All of the components depend on Kerberos user authentication from within the local network (Steiner, Neuman, & Schiller, 1988).

Figure 4.1: PyAres Architecture of the core components. The data retriever accesses the ARES database, enabling data retrieval to a local machine, or to retrieve data to the Spark job. The job submitter defines and submits Spark jobs to the YARN scheduler to run on the cluster. Job results get persisted in HDFS and can be accessed on a local machine via the result retriever.
4.2.1 Data Retriever

The current machine learning workflow used by the DATO group involves data access on their local machines via an export of MUST data to a local database or files. The PyAres data retriever component is a Python refactoring of the ARES data provision API, and connects to the ARES databases to retrieve, or transport, the specified data. This data is either transported directly to the user’s machine, or first moves to a cluster node where it is processed as part of a running Spark job. The data retriever component uses a configuration file that defines all the needed credentials and nameservices to connect to the proper ARES instance. The data retriever component has the following workflow to return the data to the machine sending the query:

- Connect to the ARES MariaDB to retrieve the parameter ID of the requested telemetry parameter.
- Translate request for parameter samples to bytebuffer keys for the HBase scan, based on parameter ID and timestamps.
- Scan HBase based on the bytebuffer keys and return the appropriate sample data.
- Decode the protocolbuffer stored data into parameter samples objects.
- Return the requested data as parameter sample objects or further unpacked into a pandas dataframe.
Figure 4.2 shows the architecture for the interaction between the different components of the data retriever. The configuration factory is used to create a generic interface that can be used by all core components to interact with the configuration file. All the other components can then be used to make the connections to the data sources and translate the retrieved data points into telemetry samples.

**Design Considerations**

Section 2.2 described the architecture of the ARES database. The data retriever component needs to be able to interact with the database in order to correctly give access to the data by either the users or the nodes during the job processing. A number of Python libraries were picked to aid in this interaction.

**PyMySQL** To make the connection with the MariaDB instance PyAres makes use of PyMySQL version 0.8.0. The consideration for PyMySQL was made based on the documentation of MariaDB itself, which recommends the usage of PyMySQL (MariaDB, 2019). One of the disadvantages of PyMySQL is that it is a library that needs to be installed separately, it is not part of the Python distribution as a built-in package, such as SQLite3. However, in contrast to SQLite3, PyMySQL is a much more mature library fit for operational deployment, based on its support for user management, which is something that SQLite3 does not have (Python Software Foundation, 2019).

**Happybase** PyAres depends on Happybase version 1.1.0 to communicate with HBase. Happybase needs a deployed Thrift server on the cluster where the ARES database instance is running for the communication. At the time of implementing, there was no satisfactory alternative interface for PySpark and HBase. The advantage of using Happybase is that it works intuitively, queries are easy to define. Communication via Thrift is language agnostic and allows for data retrieval to Python and PySpark (Apache HBase Team, 2019) without having to manually set up all the communications in Python. The disadvantage is the communication via Thrift, which causes overhead. This overhead stems from marshalling costs, where the Thrift server compatible with HBase uses a binary protocol to communicate. The search query string needs to be translated to binary to retrieve the appropriate data, and the retrieved data needs to be translated from binary to string to be readable by Python (Apache Thrift Team, 2019). After this, the ARES data is packed in a protocol buffer, which is also a binary format that needs to be unpacked.

### 4.2.2 Job Submitter

The ARES cluster is a Hadoop based cluster with a Spark instance for job processing. A PyAres job uses the data retriever functionality for data access inside the job on the Spark executor side. Figure 4.3 captures the architecture of the PyAres library. The configuration needed by the job submitter component in order to access the YARN instance is pulled from the configuration factory.
Spark was originally implemented with Scala (Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., Stoica, I., 2012), which is a JVM language, so Spark compatibility with Python meant translating Python objects and data types to JVM compatible types. Figure 4.4 shows the architecture of this implementation as done by Apache. To facilitate Spark compatibility with Python, CPython was used, which is the C implementation of Python. Using CPython meant that the Spark job executing needed to be decoupled from in line code execution. This is due to the fact that the CPython enables a Global Interpreter Lock, locking the execution to a single thread, because CPython’s memory management is not thread-safe (Wouters, 2017). Practically speaking, this means that a PySpark job needs to be defined in a script that gets submitted to the cluster via a Command Line Interface (CLI).

For the PyAres implementation this means that jobs have to be executed in two steps, where a script that defines the tasks that Spark needs to execute is created separately from the script that submits the job and returns job feedback to the user. The ares job component consists of three sub-components that take care of job definition, job submission and job feedback.

**Design Considerations**

For the design of the components responsible for the job definition, submission and feedback there were a small number design considerations to make. Due to the fact that the submission of jobs needed usage of the spark-submit CLI, this meant that the PyAres library needed to be deployed on an ARES configured gateway node, as mentioned in section 4.2. This also meant that the YARN CLI was available so instead of adding another library to the PyAres dependencies, the choice was made to leverage the output of the YARN CLI to provide the user feedback.
Figure 4.4: PySpark Data Flow architecture. The locally initialized Spark Context translates Python data types through a Py4J socket to JVM compatible data types, which is launched on initialization of the Spark Context. On the cluster, the JVM types are transformed to Python through a pipe that sends a pickle with user code and data to be processed to the Python sub-processes. Source: (Apache Spark Team, 2016)

Job Definition

The job definition is the sub-component that translates Spark specific implementation details, needed for correct data retrieval and processing, to a more user friendly and Spark agnostic interface. This way the data scientist can perform their normal data pre-processing, as they might be used to from other data science libraries such a pandas. For example, Spark 1.6 has no native implementation for data re-sampling with an aggregating function and interpolation of missing data. Whereas the pandas implementation for these operations is a simple one-liner, the Spark implementation required a much more low level implementation of these algorithms and the definition of multiple functions to handle this. Just to illustrate, to re-sample to 1 hour using mean for the aggregation, after which a linear interpolation of the missing data takes place, the pandas equivalent would be:

```python
1 df = df.resample('1h').mean().interpolate(method='linear')
```

The job definition depends on the job helper, which implements the retrieval of the samples and a bunch of built-in checks to make sure the data can be retrieved and the retrieval can be run in parallel on the nodes. Two types of jobs can be defined: a statistics job and job based on User Defined Functions (UDF).

Statistics Job The statistics job is based on existing ARES functionality. ARES offers scheduled statistics jobs for predefined resolutions (5m, 1h, 1d) that calculate nine statistics by default (min, max, average, median, count, kurtosis, skewness, variance, standard deviation), as defined in the Software
Requirement Specification SRS for scheduled statistics jobs in ARES (ARES team, 2017b). The DATO group would like more control than what this ARES functionality offered and requested the possibility to calculate statistics on the fly for custom resolutions.

**UDF Job**  The UDF job is based on the possibility to do data pre-processing as one would do using pandas. The DATO group has stated the desire to be able to parse different data types (int, float, string) to a new format. An example, which is more elaborately explained in section 5.2, would be to transform the the degrees of an angle of a spacecraft orientation to radians. The user defined functions mentioned in this context are user defined scalar functions. They are n-to-n operations and operate on the rows of the input column, e.g. the individual attributes of the specific record. The user can define functions that calculate new columns in an element wise way. One of the obstacles in the Python implementation of these Spark UDFs in Spark 1.6, is that Spark expects a predefined output type to be declared. But when the user has the freedom to create their own UDFs, the output type cannot be anticipated beforehand. To give PyAres information about the return type of the UDFs, the user has to use annotation types to pre-define the return type. This way, PyAres can use the function signature to extract the return type and cast the proper type on the new column, e.g.

```python
@overloads
def project_angles(param) -> float:
    return float(max(0, np.cos(np.radians(param))))
```

**Job Execution**

The job execution of the job submitter component is a very straightforward sub-component. It offers functionality to upload the script to the cluster onto the YARN resource manager where the rest of the execution takes place. As mentioned before, PySpark 1.6 does not offer any functionality of inline job execution due to Python language restrictions, so the job execution functionality composes a string for the CLI, which gets executed with the Python built-in OS library. The command it composes can be found in figure 4.5. The main components of the command is to set the proper configuration parameters as read from the configuration file, and to upload the script and its dependencies to the cluster.

The script depends on a number of files and an archive with a portable Python virtual environment. This Python virtual environment is attached as a zip archive. The archive needs to be attached because the ARES nodes are using a Suse Linux Enterprise Server (SLES) 12 baseline (which runs Python 2.7 as its default) and cannot be modified to use Python 3, as defined by the baseline deployment strategy at ESOC. Since Python 3 is needed to run Spark and the other Python components, the archive is included, unpacked as part of the container where the Python 3 binary is used to run the code. Other files submitted to the cluster include the configuration file needed for the rest of the configuration on the nodes, and some external Python variables, that are declared as part of the submit script, included as a pickle.
Figure 4.5: PySpark submit command code. This command is composed and executed as part of the job execution functionality.

The files and archives that are needed for the script to properly execute (pickle, configuration, and virtual environment archive) can be renamed and specified with the # sign, similar as in Hadoop. This makes sure that the filenames are persisted within the cluster, and transformed to the HDFS namespace so that everything can be accessed from the container running the job (Apache Spark Team, 2019).

### Job Feedback

Once the job has been successfully submitted to the YARN resource manager, we can send requests to YARN asking for the job status based on the application id. This functionality was added for user feedback and convenience, to keep track of the status of the job and whether the job ran into any problems. There are two functions defined for job feedback, to get the job status and to get the job logs.

#### Job Status

The status of the job is returned to the user as a dictionary with the state, final state and progress. This information can be used to define a callback function that waits for the job to be finished and to automatically retrieve the results on successful completion.

```python
1 cmd = ('spark-submit
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  config_path,
13    data_path+venvs,
14    str(wait_completion).lower(),
15    str(spark_conf["driver_memory"]),
16    str(spark_conf["executor_memory"]),
17    int(spark_conf["executors"])
18  )
19  cmd_out = subprocess.check_output(cmd, shell=True, stderr=subprocess.STDOUT)
20  cmd_out = cmd_out.decode("utf-8")
```

Esther Kok - s4140389 23 January 16, 2020
Job Logs  During the job processing specific PyAres log entries are created to keep track of anything that went wrong during the job, e.g. parameters returned no data within the specified time-period, the user defined functions could not be executed with the parameter types, etc. Based on this feedback the user can adapt their work or look into problems, such as data missing from ARES that should be expected to be there.

4.2.3 Result Retriever

The final component is the result retriever component. It is a very straightforward function that pulls the results of the job from HDFS, where they are persisted, and transforms it into a pandas dataframe to be returned to user on their local machine. As can be seen from figure 4.6, the architecture is very simple, there is just an interface with the configuration factory to know which location and credentials are needed to find the HDFS stored files.

Design Considerations

The files in HDFS that are part of the result are stored in Parquet. Parquet is a file format defined by Apache that compresses columnar data for the Hadoop ecosystem. It is designed for efficient compression and decoding, so the data retriever, depending on the volume of the data, is quite fast. Spark offers built-in functionality to write results to Parquet, so the resulting Parquet files are written to HDFS to a location specified in the configuration file, based on the user, e.g. the mission using the PyAres instance, user access and namespace (Apache Parquet Team, 2019).

Although Parquet has the advantages of efficient compression and decoding, the consideration was made to create a separate HBase instance for persisting the results in an HBase format, because then the data would be persisted in an elegant distributed database instead of distributed files. However, there were a number of constraints that drove the decision to write the results into HDFS. These constraints will be discussed a bit more in depth in section 4.4.

For the retrieval of the results from HDFS, PyAres depends on pyarrow version 0.9.0. Pyarrow is an Apache supported library for accessing files on HDFS and offers functionality for unpacking Parquet files (Apache Software Foundation, 2019). There were no other libraries considered because of these advantages.
Figure 4.6: PyAres Architecture of the result retrieval component. Results are stored in Parquet and returned to the user as a pandas dataframe.

4.3 Operational Deployment

The PyAres library is deployed on the operational cluster as a Docker container running a Jupyter Notebook. The consideration for this deployment was the fact that ARES is deployed on SLES12 machines, which means that PyAres needed to be SLES12 compatible as well. By deploying the library as part of a Docker container, the user can interface with the library and a familiar environment, e.g. Jupyter Notebook. The user can use this environment to develop a data analytics prototype which can be directly submitted to the ARES cluster, without worrying about installations and configurations. The container can be considered a trusted environment. The decision was made to deploy a container for each mission that wanted to use the PyAres functionality to make the configuration easy. This gave each mission their own location for data persistence, both in terms of submitted jobs and job result data, in order to keep data from different missions separate.

4.4 Limitations

Some of the design choices were not made because they were the best option available, but because of a number of constraints that arise due to the type of environment that exists at ESOC. One of the major constraints in implementing the best solution possible, is the usage of software baselines. Limiting design choices related to results persistence, were caused by time constraints and enforced user administration. The limitations and corresponding costs were deemed acceptable for a first version of this prototype, but should be addressed as part of future work.

4.4.1 Software Baselines

At ESOC, software baselines are meant to guarantee operational functionality of newly implemented software in concordance with other systems already in place. The companies designing the software all work within the same baseline versions. Not only facilitates this software interaction, but it also makes testing the new software within this environment more straightforward. Missions who deploy their spacecraft can be guaranteed a solution that has been thoroughly tested and functions for the duration of the mission, as well as allows newer software to function within that environment.
Production grade software at ESOC all functions on the SLES12 baseline. This causes constraints on the versions of other software and programming languages. In terms of ARES, this puts constraints on the CDH version (5.11), the Spark version (1.6), and the HBase version (1.2) (Cloudera, 2019a). This limits the usage of newer versions of Spark, especially Spark 2.x, which has a more optimized Python interface (Apache Spark Team, 2018). Utilizing Spark 1.6 means taking a cut in performance as an acceptable loss in this context.

It also meant that the Python version that was available for the development of PyAres was limited. The Python version SLES12 defaults to is Python 2.7, but as of January 1st 2020, it is no longer supported (Python Software Foundation, 2020b). Python 3.4 is the SLES12 supported version of Python 3, and only available via additional package installation. However, Python 3.4 has also deprecated; since March 2019 it is marked as end of life (Python Software Foundation, 2020a). This not only means that the security can no longer be guaranteed for these versions, but also that any libraries used as dependencies in software written in these Python versions, have the same issue. PyAres suffered no severe limitations in its dependencies, so this was deemed an acceptable cost.

One of the more problematic things associated with the PyAres prototype implementation is the slow data access performance due to data marshalling via Thrift via Happybase. Section 4.2.1 mentions these problems in a bit more detail, but the main issue is caused by transporting data from HBase to Python via Happybase and Thrift. Upgrading the SLES baseline would make the infrastructure compatible with a newer version of Cloudera CDH. Cloudera CDH version 6.3 comes packaged with Spark 2.4 and HBase 2.1.4, for which there is a Python compatible HBaseConnector available that allows for HBase queries and filters to retrieve the data (Cloudera, 2019b). This should impact performance due to a circumvention of marshalling costs associated with Happybase and Thrift.

### 4.4.2 Result Persistence

For the first version of the prototype the choice was made to persist the results from the PyAres Spark jobs on HDFS. Persisting the results this way, instead of a separate HBase instance, comes with a cost in terms of read/write speed. HBase contains an in-memory processing engine to optimize I/O, whereas HDFS uses MapReduce processing, so no in-memory processing, to facilitate this (Apache HBase team, 2018). The cost was deemed acceptable for the version 1.0 PyAres prototype, due to a number of reasons.

First, there was a time constraint in the delivery of the first version of the prototype, so writing to HDFS was the most straightforward way to implement this feature. As mentioned before in section 4.2.3, the consideration was made to use a HBase instance for job result persistence, but was not further explored due to these time constraints.

Second, the write access itself was an issue. User are authenticated using Kerberos for access to ARES. Production grade implementations should not have overall write access on the HBase instance, so write access to the HDFS file system in a specific folder was more straightforward to implement as well. Having specific users with authentication to write job results to a PyAres HBase instance would be interesting to explore further, but was dropped again due to time constraints.
Chapter 5

Use Cases

5.1 Use Case Goals

The overarching goals of the use cases is to check how well existing use cases translate to the new implementation, and to check how well it deals with large volumes of data. The new implementation boasts usability for users unfamiliar with the architecture, and an decrease in use case completion time, so this needs to be evaluated. This translates to the research questions

How well does the current data layout generalize for algorithms beyond DrMUST and how can it be improved upon?

and

What is the cost of (not) adapting client code to the big data platform, e.g. machine learning code in Python?

Two use cases were defined to evaluate PyARES; MEX and Aeolus. The reason for picking MEX as a use case candidate for PyAres was to evaluate how well the first implementation can be used to implement use cases. MEX is an existing use case that has been analyzed before by the DATO team, and therefor has well defined data pre-processing (Petkovic, M., Boumghar, R., Breskvar, M., Dzeroski, S., Kocev, D., Levatic, J., Lucas, L., Osojnik, A., Zenko, B., Simidjievski, N., 2018). Aeolus was chosen as a use case candidate in order to evaluate the usability of the PyAres implementation for the generation of progress reports. Since a regular desktop should not be able to handle the amount of parameter samples, and the anecdotal evidence pointed to a performance of a work day to generate a progress report, the performance of the cluster deployed PyAres implementation would be the most important indicator for the libraries usability in this and future similar use cases.
5.2 Mars Express

5.2.1 Mission

The Mars Express (MEX) orbiter was launched on June 2\textsuperscript{nd} 2003 and arrived at Mars on December 25\textsuperscript{th} of the same year. It carried a lander payload called Beagle 2 which was jettisoned on December 19\textsuperscript{th} 2003 and would have entered atmosphere on December 25\textsuperscript{th}, but the lander was declared lost on February 6\textsuperscript{th} 2004 after contact could not be established (ESA, 2018j). The MEX orbiter was still operational in orbit and still is to this day.

The MEX orbiter was designed to study all aspects of Mars; atmosphere, climate, mineralogy, geology etc. It follows an elliptical orbit for different kind of observations. A satellite around Earth often follows a circular shaped orbit, because the observations have to be comparable, e.g. satellite pictures always need to be under the same time and distance conditions. Since the MEX is a science mission, it was designed to get as many different measurements possible to get a full picture of the observed celestial body (ESA, 2018k).

Since its arrival in Mars’ orbit and the start of its scientific operations, the MEX orbiter has made quite a few interesting discoveries about Earths planetary neighbor. The most high profile discoveries focus on water and include the discovery of subsurface layers of water ice, and hydrated minerals, suggesting that the planet used to be much wetter than it is today (ESA, 2018k). That the MEX is still a very relevant spacecraft is proven by its recent discovery of water directly under the surface of the south pole, thanks to its still operational radar sensors (ESA, 2018h).

Its planned mission lifetime of one Martian year (687 Earth days) has long since passed and the MEX orbiter has been operational for fifteen years now. Because of its older age, the orbiter will, at a certain point, encounter problems with failing subsystems. Recently it received a software update to ensure longevity and operational reliability of certain subsystems beyond 2020 (ESA, 2018i). One of the things that cannot easily be fixed with a software update are its dwindling power supplies due to degradation of the solar panels. The spacecraft uses electrical power coming from the solar panels or batteries during eclipses. These supply the platform units and the thermal subsystem, which keeps the entire spacecraft within a temperature range. The remaining power can be used by the payloads to do science operations. The software can help account for the lower supplies by making efficient use of power supplies, and that is what ESA’s 2017 machine learning competition aimed for; to make a prediction of the required future power consumption (ESA, 2018f; Petkovic, M., Boumghar, R., Breskvar, M., Dzeroski, S., Kocev, D., Levatic, J., Lucas, L., Osojnik, A., Zenko, B., Simidjievski, N., 2018).
5.2.2 MEX Challenge

In 2017 ESA published a machine learning competition with the aim to predict the required power for thermal lines, so an estimation of the required power for the future can be made. The power that the thermal subsystem will consume is dependent on the heaters, whose behavior depends both on external (e.g. sun distance, solar aspect angle) and internal heat sources (e.g. units’ status, payloads’ status). Different systems in the spacecraft need different temperatures to operate at optimal capacity, e.g. electronics need room temperature where imaging sensors need a low temperature (ESA, 2018f; Petkovic, M., Boumghar, R., Breskvar, M., Dzeroski, S., Kocev, D., Levatic, J., Lucas, L., Osojnik, A., Zenko, B., Simidjievski, N., 2018). The published train data consisted of different measurements along the entire spacecraft for three Martian years with power output for 32 thermal lines. The test data contained the fourth Martian year. The challenge was to predict power requirement for the fourth Martian year at a 1h resolution for all 32 thermal lines.

The data that was provided at a 1h resolution consisted of:

- **Training Data**
  - SAAF: solar aspect angles
  - DMOP: detailed mission operations plan/commands
  - FTL: flight dynamics with pointing events
  - LTDATA: long term data (like constants, sun-mars-earth-angle, eclipse durations etc.)
  - EVTF: miscellaneous events (like time spend in penumbra/umbra)
  - Targets: 32 features for thermal power lines with measured electric current

- **Test Data**
  - SAAF: solar aspect angles
  - DMOP: detailed mission operations plan/commands
  - FTL: flight dynamics with pointing events
  - LTDATA: long term data
  - EVTF: miscellaneous events

The prediction of the thermal lines means the prediction of the telemetry parameters, while the rest of the data used for the training consists of event data for the EVTF data, telecommand data for the DMOP and FTL, and spacecraft specifications for the SAAF data and LTDATA.

From the outcome of the challenge the conclusion could be made that one of the most important features for predicting the thermal power requirements are the solar aspect angles, or the orientation of the spacecraft in respect to the sun, in combination with whether the spacecraft resided in (half) shadow or not. Figure 5.1 shows a schematic representation of these shadow events, as well as the elliptical orbit the spacecraft follows. The intuition makes sense, the spacecraft would optimize the utilization of its power in such a manner that the power input is biggest at the same moment to prevent overloading the battery.

Esther Kok - s4140389 29 January 16, 2020
Figure 5.1: MEX orbit around Mars with schematic representation of different shadows cast by the sun on the spacecraft. Marked are the occurrences of the different events in the data relative to the location of the spacecraft in the shadow. (ESA, 2016)

Figure 5.2: Illustration of the MEX orbiter and the axes x, y and z, which correspond to front, left and up. These axes can be used to derive back, right and down. $\alpha x$ is the solar aspect angle of the front side, i.e., the angle between the normal $\hat{n}x$ and the Sun-MEX line. $\hat{n}p$ marks the normal of the panels (Petkovic, M., Boumghar, R., Breskvar, M., Dzeroski, S., Kocev, D., Levatic, J., Lucas, L., Osojnik, A., Zenko, B., Simidjievska, N., 2018).
5.2.3 PyAres Implementation

To process the data from the MEX challenge with PyAres, the data was transformed to be compatible with ARES’ data importers and stored in an ARES HBase instance. The ARES instance was deployed on the prototyping cluster, where access to operational data and infrastructure is limited, to make sure the operational cluster was not disturbed, and the early prototype could be tested. A job was submitted to the prototype cluster that carries out the task of transforming the solar aspect angles (sa, sx, sy, sz) into radii for the three years of data imported into ARES with a 1h resolution. The result of the calculations were seven columns (alpha, right, left, up, down, front, back). Figure 5.2 marks the location on the spacecraft to which these columns correspond. The sample count of the data retrieval for each solar aspect angle during these three years was around 3,500,000.

The measurements of the performance are for two PyAres components; local data retrieval and cluster calculations. With local data retrieval the original ARES data is transported from HBase to a client machine, which enables individual users data access to the ARES HBase data lakes. Cluster calculations can be defined as the importation of the ARES HBase data into a Spark process which carries out some calculations on the Spark enabled cluster nodes. The Spark job completion time is measured by taking the time between job start and job finished as written in the log files. Because PyAres for MEX runs on the prototype cluster, not on the production cluster, the absolute time may only be indicative of the scalability of the implementation.

Transporting the ARES data to a client machine is considered not suited for large volumes of data, because it is limited to the memory of the machine, so it should only be considered for prototyping of methods that will be run on the cluster for the data pre-processing. However, for the MEX use case, the sample rate is low enough that the total number of returned samples for the entire use case does not exceed 3,500,000, making the retrieval to the client feasible. The client in this case is a virtual machine with similar specifications to the cluster, see table 5.1. Comparing performance of the client versus the cluster is not an entirely fair comparison, because the cluster has severely limited resources as part of the configuration. So even though the machine has the resources available as described in table 5.1, the configuration limits the available RAM to 512MB. The number of executors available to the job of cluster data processing is 5.

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Speed</td>
<td>2.50 GHz</td>
<td>2.50 GHz</td>
</tr>
<tr>
<td>CPU No. Cores</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
<td>4GB</td>
</tr>
<tr>
<td>No. Nodes</td>
<td>N.A.</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.1: Specifications of the machines for the client and the cluster. The cluster has a number of nodes with the same specifications, where the client is a single virtual machine.
Data Retrieval Performance

One of the biggest influences on the performance is the data retrieval via Happybase, as discussed in sections 4.2.1 and 4.4. For distributed data retrieval, transporting the data to the executor nodes for further processing, each connection needs to be initialized in the individual executor node, because the connection is not serializable. To test in what manner Happybase parameters influence the data retrieval, a series of tests was performed with different values for the parameters Connection Pool Size (1, 10, 25, 50, 100) and Scan Buffer Size (500, 1000, 2500, 5000, 10000, 50000, 100000, 500000). Happybase did not offer many parameters to tweak, where the documentation mentioned one other parameter, Protocol, but the latest version of Happybase did not recognize this parameter when trying to tweak it.

One year of data, around 700,000 samples, was retrieved for two iterations of the same experiment; one iteration to measure data retrieval to a client machine, and the other to measure performance for retrieval to the nodes as part of a Spark job. Figure 5.3 shows the performance in data retrieval for these two iterations, where the top graph shows the performance for the cluster side data retrieval, and the bottom graph for the client machine data retrieval.

The results from the test, as shown in figure 5.3, show that there is a minimal effect by the the amount of connections in the pool and the batch size for the data transported to the client machine after a certain threshold has passed. There seems to be little effect from the scan buffer size on the data transportation to the nodes. This can be explained by the fact that PyAres processes and transports data on the nodes in small partitions. Each partition consists of data for one parameter with a resolution of five minutes, meaning that the batches it processes and transports are very small. The data transportation to the client machine does not have this resolution limitation so is transported in larger batches. The connection pool size does not influence the performance on the cluster either. This can be attributed to the implementation of PyAres again, where partitions of the data retrieval tasks get executed on the node based on one HBase connection, so only one connection in the connection pool is utilized.

Use Case Implementation Performance

Based on the results from the data retrieval test, the configuration for the data retrieval was chosen; Scan Buffer Size 500,000 because the transportation to the client machine was optimal for this number, and Connection Pool Size 10, because the number seems to be arbitrary. Based on this configuration another test was performed where the MEX use case implementation was tested on different time period lengths (one day, one week, one month, one year and four years) for the same resolution (one hour). This experiment resulted in a certain number of samples being read from HBase. Figure 5.4 shows that the retrieved number of samples for the each of the time periods is: {one day: 1969, one week: 14449, one month: 67541, one year 664436, four years 3427478}.

The results, as seen in figure 5.4, show that even though the performance is limited by the resources on the cluster side, the PyAres solution does scale, whereas the client side processing remains more
Figure 5.3: Performance of the data transportation to the prototype cluster (top) and a client machine (bottom), measured in seconds. The size of the parameter Scan Buffer Size (500, 1000, 2500, 5000, 10000, 50000, 100000, 500000), e.g. the number of samples that can be retrieved per scan, is plotted against the Connection Pool Size (1, 10, 25, 50, 100), e.g. the number of connections to HBase the application has available for retrieving data.
or less stable. Smaller cluster side jobs suffer from a lot of initialization overhead, but the overhead per sample becomes smaller for larger jobs. While the performance for the available four years of data does not drop below the level of the client side processing, the solution scales and should be evaluated on production grade architecture with a use case with a higher sample rate. The results of also raise the question whether a modern desktop is strong enough these days that it can handle most of the use cases that this prototype would be addressing.

5.3 Aeolus

5.3.1 Mission

Aeolus, named after the god from Greek mythology who was assigned as keeper of the winds by Zeus, is an Earth Observation mission that uses very sophisticated Doppler wind lidar technology to accurately map wind profiles in almost real-time. Aeolus’ payload, the Atmospheric Laser Doppler Instrument or Aladin for short, uses light scattering and the Doppler Effect to acquire data on wind (ESA, 2018e, 2018g). Aeolus was launched on August 22\textsuperscript{nd} 2018 and is being operated from ESOC (ESA, 2018b). Aeolus is one of the missions whose data resides on the ARES production cluster and has a high telemetry sample rate, therefore it makes for a good candidate to measure PyAres performance on the operational infrastructure.

![Figure 5.4: Scalability of processing time of MEX use case. The two series ‘client’ and ‘cluster’, point to the different implementations of the use case, where the ‘client’ series signifies the data transportation to a client machine after which the data is processed, and the ‘cluster’ series points to the Spark job submitted to the prototype cluster. Data is processed to a resolution of one hour for five time periods (one day, one week, one month, one year and four years).](image)

Esther Kok - s4140389 34 January 16, 2020
5.3.2 Spacecraft Status Report

One of the Aeolus Spacecraft Controllers (SpaCon) got involved in the PyAres project because they expressed an interest in optimizing their workflow. In the discussions described in section 3.4, this particular SpaCon expressed that one of their tasks involved the generation of weekly and monthly operational mission status reports. The generation of these reports involved some manual effort, due to the data not being accessible via Python. The SpaCon wished to use a Jupyter Notebook to generate these reports. However, analyzing the data for the graphs required them to select the appropriate time periods for each parameter, export the required sample data via MUST to a CSV, read the CSV with Pandas and calculate the statistics for each of these parameters with a certain time resolution. Figure 5.5 shows an example of a generated graph that might be included in these status reports, in this case the laser power telemeter for the Aeolus payload plotted with five minute resolution for one week. A report might include somewhere between ten to a hundred parameters, depending on the events that occurred during the time period covered by the report, though the laser power of the payload is one of the most prominent ones due to the science mission of the spacecraft. The generation of such a report costs the respective SpaCon up to a full work day worth of effort. After discussing the status reports with the SpaCon, the decision was made to evaluate the scalability of the performance of PyAres on the production cluster. The production cluster offers the architecture to properly assess the implementation, because it has many more available resources and is the cluster that is used for all the big data processing.

5.3.3 PyAres Implementation Performance

The evaluation of the performance of retrieval of data to a local machine is a bit hard to measure empirically, because performance of the old situation depends on anecdotal evidence, but in order to quantify the results of the data retrieval a similar test as with the MEX was conceived where the data for a week was retrieved for a set of ten to a hundred telemetry parameters, as deemed useful for the status report. However, when retrieving the data to the client side, it became apparent that retrieving data over 10 parameters to the client side would overload the client to be point where it became non responsive. Therefore, in the end, it was decided to abandon this part of the test, and to go straight toward evaluating a cluster side processing job for this particular use case. The production cluster offers the same specifications as the prototype cluster in terms of available resources, but the number of executors per job is much higher, e.g. 50 executors per job instead of 5.

The Aeolus use case job consisted of a so called statistics job, which calculates all nine possible statistics as described in section 4.2.2, for each involved parameter. The statistics were calculated over different time periods (one day, one week, one month), for certain amounts of parameters (10, 25, 50, 75, 100). The parameters included in the calculations were to be randomly selected from the available parameters to account for different HBase region servers being involved in the data retrieval, but the subsets were always the same for each experiment so that the processed number of parameter samples is only dependant on the time period. Table 5.2 shows the number of parameter samples involved in each of the jobs. The resolution of the calculated statistics was five minutes for each of these jobs, defined as relevant by the SpaCon.
The performance was measured over five iterations of the same statistics job for each of the parameter subsets, because the duration of the job is dependant on cluster resource availability. The outcome of the five iterations of the same job was used to calculate some descriptive statistics (mean, min, 25%, 50%, 75%, max) in order to give a more balanced overview of the performance during different periods of cluster activity. As can be seen in figure 5.6, the cluster resource availability impacts the performance, but not that much. For a job of one hundred parameters over a period of one month, the average performance of 1818 seconds, or 30 minutes and 18 seconds, is much more efficient than the current workflow of the SpaCon.
Table 5.2: For $n$ parameters with a certain time period, the job resulted in an above described number of processed samples.

<table>
<thead>
<tr>
<th>$n$</th>
<th>1 day</th>
<th>1 week</th>
<th>1 month</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>417610</td>
<td>2897434</td>
<td>12501864</td>
</tr>
<tr>
<td>25</td>
<td>905912</td>
<td>6285260</td>
<td>27119676</td>
</tr>
<tr>
<td>50</td>
<td>2135816</td>
<td>14824707</td>
<td>63964376</td>
</tr>
<tr>
<td>75</td>
<td>3087449</td>
<td>21417556</td>
<td>92420017</td>
</tr>
<tr>
<td>100</td>
<td>3784415</td>
<td>24942920</td>
<td>107766816</td>
</tr>
</tbody>
</table>

Figure 5.6: Performance of the data retrieval to the cluster, measured in seconds.
Chapter 6

Discussion

6.1 Research Questions

The main goal of the work described in this thesis was to identify the reasons for the machine learning engineers from the DATO group not using the local big data lakes, and to improve on that situation. In this chapter the main research questions shall be discussed and concluded, as well as follow-up points for future work. The main questions are:

- How are the machine learning engineers at ESOC using the big data lakes?
- What are the bottlenecks in data access for the machine learning engineers?
- What are the bottlenecks in result persistence?
- How well does the current data layout generalize for algorithms beyond MUST and can it be improved upon?
- What is the cost of (not) adapting client code to the big data platform, e.g. machine learning code in Python?
- How can we make the ARES interface usable for the DATO group at ESOC?
- Can we improve on data pre-processing using the ARES interface?
6.2 Bottlenecks

Chapter 3 offered a method to identify the different bottlenecks in access to data lakes present at ESOC, as well as the persistence and reusability of results. In summary, the discussion outcomes identified in that chapter all relate in some way to a lack of Python interface with the ARES data. The engineers are accessing the data stored in the data lakes via a MUST data export, after which they can process the data in their usual workflow via Jupyter notebook. The main blocking points are; a lack of direct access via Python, a lack of appropriate centralized storage of the result from the data processing, and a lack of a way to process the larger volumes of data ARES offers.

These blocking points influenced the way that the PyAres prototype was developed and deployed. After running the two different use cases with PyAres, it can be concluded that the prototype, deployed via Docker on Jupyter notebook, offers a workable interface to the production deployed ARES data and processing resources, without complicating the process. The user can gain access to the web interface of the Jupyter notebook, therefore having ease of access and no installation or configuration tasks, and can start working with the library and other data science libraries straight away.

6.3 PyAres Limitations

Section 4.4 discusses several limitations associated with the implementation of PyAres. While some of these limitations are related to administrative restrictions, and are therefore considered acceptable limitations, they still warrant some discussion as part of this chapter.

6.3.1 Happybase, Thrift and Protobuf

The design considerations described in section 4.2.1 point out that Happybase and Thrift might not be the best choice in terms of performance, but were chosen based on administrative reasons. In the end, there was no satisfactory interface for Python and HBase due to ESOC enforced software baselines. The consequence of this decision is quite clear in the slow performance caused by marshalling costs.

The combination of translating the query string to binary, the result binary to string, after which the result content needs decoding from binary string to objects via Protobuf, is detrimental to the performance of the result retrieval and subsequent data processing. In contrast, because Apache HBase is Java compatible, some of these marshalling costs will not be there and the HBase data can be indirectly translated with the Protobuf decoder, skipping some of the overhead. As mentioned in section 4.4, the upgrade of the baseline to a SLES15 baseline should make it possible to use the Python compatible HBaseConnector, which would be interesting to explore in term of performance improvement.

To measure the difference between Java and Python marshaling costs, a small experiment can be devised that would measure the HBase data read time for both languages. The current ARES HBase
instance could be deployed in a Docker container and accessed via Java or Python, reading the same data samples. Deploying the HBase SLES15 version (e.g. HBase version 2.4.1) could also be used to measure the difference in performance between Java, Python Happybase and Python HBaseConnector.

Another consideration that can be made relating to the marshaling costs, is to store the data outside of the Protobuf. In terms of storage, the encoding of the data in a Protobuf has some advantages, as it takes less space to store. Compared to a format like JSON, Protobuf encoded data takes up less space (Nils Magnus, 2020). However, it might be more costly to store the data in Protobuf in the end due to the marshaling costs involved when people want to access to the data. In total the ARES data takes up roughly 15TB. Even if the data encoding in Protobuf is very efficient, 25% of the raw data, the total amount of storage needed would still be within acceptable cost limits. It would be very interesting to run a small experiment to measure the impact of Protobuf on data storage and marshaling costs.

One of the other possible causes of the overhead from Happybase and Thrift, is the fact that the cluster Thrift server deployed on the cluster is running on nodes that have multiple roles, whereas the recommendation is to run Thrift on its own dedicated node (Cloudera, 2019c).

6.3.2 HDFS

The results from the data processing are written to HDFS as Parquet files. Although Parquet has the advantages of efficient compression and decoding, the consideration should be made to create a separate HBase instance for result persistence. That way the data would be persisted in an elegant distributed database instead of distributed files. However, due to multiple constraints, described in section 4.4, the choice was made for storage in HDFS. Of course, this choice is not without its consequences. Even though the user does not notice this, because it is hidden behind the PyAres API, the result persistence is not optimal in terms of disk I/O, e.g. read and write speed. Future work should include an investigation into using a separate HBase instance per mission for the persistence of the results.

6.3.3 Spark

The current implementation of the ARES cluster utilizes Spark 1.6, which is an older version of Spark. This deprecated Spark version causes a small number of limitations to the current PyAres implementation and to the cluster on a whole.

As mentioned in paragraph 4.2.2, user defined functions in PyAres are scalar, element-wise. Spark 1.6 cannot deal with user defined aggregation functions, only with the built in aggregation functions, because they are optimized for parallel processing. This limitation prevents the user from coming up with their own aggregation over parameter samples for a certain resolution, for example to calculate the average radian of an angle of the spacecraft for every day.

Spark 2.x includes functionality for optimized usage of dataframe and transformations to pandas dataframe. So aside from added functionality to user defined functions, by which users can rely on
pandas functions, the optimized data mapping also influences the performance of the calculations. It enables the application of a user defined function directly onto a dataframe column, instead of using a helper function to parse the user defined function into something Spark compatible (Apache Spark Team, 2018).

In short, upgrading to Spark 2.x would benefit PyAres processing time and usability. At ESOC, any upgrade to the baseline requires rigorous testing, hence the fact that the ARES cluster still runs on a SLES12 baseline and Spark 1.6. However, the cost in time for an upgrade should not outweigh the gains in performance and functionality for the user base, as well as the gains in time in the long run.

Looking at the results from the use cases, as shown in chapter 5, the new implementation offers improvements to the workflow currently in place, despite the limitations in the performance. It has become apparent though, from the results in paragraph 5.2.3, that smaller volumes of data do not warrant writing a job for cluster side processing. Modern desktops are capable of handling the volumes of data that most data scientists would be handling in an exploratory data science task, so the usefulness of the prototype for this kind of data processing might be low. However, the library is capable of offering an interface for handling recurrent data processing tasks, for example because a mission has new data that needs to be periodically assessed such as described in the Aeolus use case (see section 5.3).

6.3.4 Generalization

The current layout of the data in ARES, especially in combination with Happybase and Thrift, is not the most efficient in terms of marshalling costs. One limiting design choice relates to the HBase row key. The database is a two part database, one that stores the metadata in a MariaDB instance, and that data is used to create a row key in order to retrieve the data via HBase scanning. This design choice stems from the ARES Software Requirement Specification (ARES team, 2017b) where the following functional requirement has been described:

It shall be known to display a list of all the parameters defined in the ARES database. For each parameter the following information shall be shown:

- Parameter Name
- Parameter Description

Because of the way HBase is structured, as a sorted map, this cannot be done efficiently, because it requires scanning the database sequentially and unpacking the keys. Hence, the choice had to be made to store this information in a separate database that contains all the metadata. This requirement causes additional overhead, on top of the other marshalling overhead, because it requires a MariaDB connection for each data retrieval request.
6.4 Future Work

The current implementation of PyAres is not the most efficient, but offers a level of data processing customization that MUST does not offer, and allows for a quicker workflow based on current discussed workflows. Based on the discussed limitation, the following improvements can be pursued as part of future work on PyAres.

**PyAres v1.0.0 Optimizations** The main optimization that needs to be investigated is the usefulness of the HBaseConnector as a replacement for Happybase and Thrift. Further investigations in this area could also include the performance when deploying the Thrift server on a dedicated cluster node in contrast to the current implementation where the node has multiple roles.

**PyAres Future Evolution** Upgrading to Spark 2.x via upgrade of the baseline will be highly beneficial to the performance, and should be a serious candidate for future evolution of PyAres. This could also open the door to further implementations on the PyAres library, for example to include tests with the Spark MLlib to offer the users an interface similar to scikit-learn algorithms. As the current prototype is built upon the Hadoop Ecosystem, it might be interesting to look into applicability of streaming for machine learning.

**Cluster Optimizations** Current limitations in the use cases could also be influenced by the available memory of the cluster. The number of executors available to the job is much bigger on the production cluster, but the memory influences the number of parameter samples that can be processed in one executor, so the implementation might benefit from more memory.

6.5 Conclusions

In short, the PyAres interface offers a new way to access data lakes at ESOC that does not involve exporting MUST data, while also giving users a way of processing larger volumes of data. The usefulness of the library is especially high for recurrent data processing jobs. With the release of version 1.0.0, the engineers in the DATO team have an interface for ARES that overcomes their described bottlenecks, and improves on the current situation. While there are still a number of limitations in place that need addressing in respect to performance, especially in lieu of the data layout not being optimal for this implementation, the prototype is usable.
6.6 Contributions

The main contribution of the work described in this thesis to ESOC is the PyAres library that offers Python users direct access to the ARES data lakes and the processing power of the ARES cluster, without having to know anything about the underlying architecture or the inner workings of Spark. It is a proof of concept to empower the user in their data analytics and a first step towards larger usage of the ARES data lakes by the missions themselves. At the moment of delivery, version 1.0.0, there still are limitations that the user needs to take into account, as mentioned above, but the overall prototype offers a stable release that can be worked with.
References

Articles


Online Resources


ESA. (2018b). About the launch. Retrieved December 1, 2018, from https://www.esa.int/OurActivities/Observing_the_Earth/Aeolus/About_the_Launch


ESA. (2018e). Introducing aeolus. Retrieved December 1, 2018, from https://www.esa.int/OurActivities/Observing_the_Earth/Aeolus/Introducing_Aeolus


ESA. (2018g). Lasers in space. Retrieved December 1, 2018, from https://www.esa.int/OurActivities/Observing_the_Earth/Aeolus/Lasers_in_space


Esther Kok - s4140389 45 January 16, 2020


Documentation

Appendices
Appendix A

PyAres API
In this API you will find examples of usage for each function, as well as a full example of usage for all the core components and the entire library.

The API is also available as a Jupyter Notebook here: PyAres_API.ipynb. Best practice is to use the notebook with nbextensions Table of Contents and Collapsible Headings.

- **PyAres**
  - pyares_config
- **Parameter Data Retrieval**
  - pyares.init_param_sampleprovider
  - pyares.init_param_sampleprovider.get_param_name_list
  - pyares.init_param_sampleprovider.get_param_metadata_df
  - pyares.init_param_sampleprovider.get_parameter_data_df
  - pyares.init_param_sampleprovider.get_parameter_data_objs
- **Parameter Data Retrieval Usage Example**
- **Job Submission**
  - pyares.init_job
  - pyares.init_job.define_udf_job
  - pyares.init_job.define_statistics_job
  - pyares.init_job.execute_job
  - pyares.init_job.get_job_status
  - pyares.init_job.get_job_logs
- **Job Submission Usage Example**
- **How To Define a UDF**
- **Result Retrieval**
  - pyares.init_jobresultretriever
  - pyares.init_jobresultretriever.get_job_result_df
- **Result Retrieval Usage Example**
- **Full Usage Example**
  - Example 1: Data Exploration Included
  - Example 2: Previous Results Only
  - Example 3: Statistics Only

---

**PyAres**

Main entrypoint into the library.

```
import pyares
```

**pyares_config**

When importing PyAres, a `PyaresConfigFactory` object is automatically created with the name `pyares_config` that allows for custom configuration file import. It defaults to handling the default configuration without interference from the user, but the function `pyares_config.set_conf('/path/to/custom_conf.ini')` allows you to set the path to a custom configuration file, which gets then included in all the initializers. A template for the configuration file can be found in the root of the repository on GitLab.

```
from pyares import pyares_config
pyares_config.set_conf('/some/path/some_conf.ini')
```

**Parameter Data Retrieval**

Functionality for data retrieval from ARES databases to the client. The ARES cluster includes a MariaDB instance that contains the metadata and an HBase instance that contain the actual (parameter) samples as protocol buffers. Data from both instances needs to be combined for successful retrieval of samples.

**pyares.init_param_sampleprovider**

Initializer for the constructor that automatically sets up the connection to HBase and other resources based on the config file.

```
import pyares as pa
data_provider = pa.init_param_sampleprovider()
```
pyares.init_param_sampleprovider.get_param_name_list

Fetch all the parameter names in the database. Not recommended for large databases.

**DEPRECATED** Recommend use of `get_param_metadata_df()`.

Return: List with parameter names

```python
import pyares as pa
data_provider = pa.init_param_sampleprovider()
data_provider.get_param_name_list()
```

pyares.init_param_sampleprovider.get_param_metadata_df

Get list of parameter metadata into a pandas dataframe. Results can be filtered to only show parameters which contain a string in any column or in a specific column. Filtering is case-insensitive. If filter is left empty, all parameters will be fetched, which is not recommended for large databases.

Return: Pandas DataFrame with columns ['NAME', 'DESCRIPTION', 'ENGVALUNIT', 'DATA_TYPE', 'RAW_DATA_TYPE']

```python
import pyares as pa
data_provider = pa.init_param_sampleprovider()
# Return all parameters
data_provider.get_param_metadata_df()
# Return parameters with column 'DATA_TYPE' containing 'UMEDIUMINT'
data_provider.get_param_metadata_df({'DATA_TYPE': 'UMEDIUMINT'})
# Return parameters with column 'NAME' containing 'NPD116' and column 'RAW_DATA_TYPE' type containing 'tiny' (both conditions need to apply)
data_provider.get_param_metadata_df({'NAME': 'NPD116', 'RAW_DATA_TYPE': 'tiny'})
# Return parameters with any column containing 'eclipse'
data_provider.get_param_metadata_df('eclipse')
```

pyares.init_param_sampleprovider.get_parameter_data_df

Get all the available samples for a given n parameter names within a given time period. By default, fetches engineering values. Raw values can be fetched by setting 'get_raw' option per parameter. Returns a Pandas DataFrame with columns based on the input parameters, UTC timestamps and values for each parameter.

Parameters:

- **param_options**: List of Strings with parameter name(s) or a list of dictionaries with keys 'name' (string) and 'get_raw' (boolean)
- **start**: Timestamp in milli- or microseconds with the start of the period
- **end**: Timestamp in milli- or microseconds with the end of the period

Return: Pandas DataFrame with columns [timestamp, Param_1, ..., Param_n]

```python
import pyares as pa
data_provider = pa.init_param_sampleprovider()
# Get raw data for both parameters
params = [{'name': 'A', 'get_raw': True}, {'name': 'B', 'get_raw': True}]
data_provider.get_parameter_data_df(params, timestamp_start, timestamp_end)
```

pyares.init_param_sampleprovider.get_parameter_data_objs

For a given n parameters in list (or string if n=1) and given start and end timestamp, get all the data into sample objects. Returns a nested iterator for the collection of samples, where each parameter sample collection is represented by a separate iterator. Samples will contain either raw or engineering values, depending on 'get_raw' option.

Parameters:
import pyares as pa

data_provider = pa.init_param_sampleprovider()

params = ['A', 'B']
timestamp_start = 1387324800000
timestamp_end = 1388584759000
samples = data_provider.get_parameter_data_objs(params, timestamp_start, timestamp_end)

# Get engineering data for parameter 'A' and raw data for parameter 'B'
params = [{'name': 'A', 'get_raw': False}, {'name': 'B', 'get_raw': True}]
samples = data_provider.get_parameter_data_objs(params, timestamp_start, timestamp_end)

import pyares as pa
from pyares import pyares_config

# Add custom configuration file
pyares_config.set_conf('/some/path/somefile.ini')

# Initialize the needed datasources.
data_provider = pa.init_param_sampleprovider()

# Get some parameters that you want to retrieval data for within a certain time period.
params = ['A', 'B']
timestamp_start = 1387324800000
timestamp_end = 1388584759000

# Get the data as a Pandas DF
df = data_provider.get_parameter_data_df(params, timestamp_start, timestamp_end)
print(df.shape)

# Or, if you prefer, get them into sample objects and get information from the objects.
# The method returns a nested generator, so you need to iterate twice.
samples = data_provider.get_parameter_data_objs(params, timestamp_start, timestamp_end)
for sample in samples:
    for s in sample:
        print(s.get_value())

# Get raw data for parameter 'A' and engineering data for parameter 'B'
params = [{'name': 'A', 'get_raw': True}, 'B']
df = data_provider.get_parameter_data_df(params, timestamp_start, timestamp_end)
print(df.shape)

---

**Job Submission**

Functionality for submission of jobs to the ARES cluster. The ARES cluster is a Hadoop cluster with a Spark instance to parallelize and therefore speed up job processing. The PyAres job uses the Data Retrieval functionality, and therefore the HBase and MariaDB instances, for data retrieval inside the job.

Spark was originally implemented with Scala (Zaharia et al., 2012), which is a JVM language, so Spark compatibility with Python meant translating Python objects and data types to JVM-compatible types (Apache, 2016). The single threaded nature of Python combined with the needed translation step results in Python job submission that is not compatible with in line code execution. Meaning that a PySpark job needs to be defined in a script that gets submitted to the cluster via command line interface. For PyAres this means that jobs have to be executed in two steps, where a script that defines the tasks that Spark needs to run is created separately from the script that submits the job and returns job feedback to the user.

To take away this complexity, the Job Submission part of the API helps with creating a `submission script` and a `definition script`. The `submission script` is the script that the user executes to submit the job to the cluster. For the sake of the API this is called the `client side script`. The `definition script` is the script that defines the job and is the script that gets submitted to the cluster by the client side script. Because it executes on the cluster, we call it the `cluster side script`. 
pyares.init_aresjob

Initializer of the constructor for Ares jobs. Automatically takes care of the needed connections to the cluster based on the config file. This initializer applies to both the client side script and the cluster side script.

```python
import pyares as pa
ares_job = pa.init_aresjob()
```

pyares.init_aresjob.define_udf_job

Use only in the cluster side script.

Define a job that can run calculations on parameter sample data in the ARES cluster. In the job definition you decide to run two types of job on two types of data.

**Job types:** User Defined Functions (UDFs) and/or Statistics job.

**Data types:** Use search data from HBase and/or persisted results from previous jobs.

**Parameters:**

- `function_dict`: Dict containing the UDFs and the column names (either parameter names or columns in previous results) to perform them on.
- `perform_search`: Bool to flag whether you want to search for parameters in the job, defined in the client side script.
- `resolution`: Can be String 5m, 30m, 1d, OR any Int in microseconds
- `previous_results`: Dict defining which job_id's and which result types you want to reuse in the job

**Return:** None

**function_dict:**

Syntax: `{key: (udf, [params])}`

- `key`: String of the name for the resulting column
- `udf`: a direct reference to the defined function, e.g. `def udf(x):
- `params`: String parameter or column names from either searched parameters or persisted results, e.g. `['A','B']`

**Important Note:** Please read the subsection about how to define a UDF for PyAres ([here](#)) to address any issues that might arise regarding the UDFs

**previous_results:**

Syntax: `{job_id: [result_type]}`

- `job_id`: String with job_id from previous results, e.g. `'application_1234567890123_1234'`
- `result_type`: One or more of result_types `['search', 'stat', 'udf']`
  - `search`: The persisted samples from the searched parameters.
  - `stat`: The result from the statistics job for a previously defined resolution.
  - `udf`: The result from the user defined functions from the submission script.

Cluster side script with UDF

```python
import pyares as pa

def udf1(x, y) -> float:
    return float(x-y)

def udf2(x) -> str:
    if x >= 100
        return '0'
    else:
        return '1'

function_dict = {'example_a': (udf1, ['A','B']),
                 'example_b': (udf2, ['A']),'example_c': (udf2, ['udf_C'])}

# We want to calculate udf2 over the results from a previous job, which we know has column 'udf_C'
previous_results = {'application_1234567890123_1234': ['udf']}

ares_job = pa.init_aresjob()

ares_job.define_udf_job(function_dict=function_dict,
                        perform_search=True,
                        resolution='5m',
                        previous_results=previous_results)
```
pyares.init_aresjob.define_statistics_job

pyares.init_aresjob.define_statistics_job(stats='all', perform_search=True, resolution='5m', previous_results=None)

Use only in the **cluster side script**.

Define a job that can run calculations on parameter sample data in the ARES cluster. In the job definition you decide to run two types of job on two types of data.

**Job types**: User Defined Functions (UDFs) and/or Statistics job.

**Data types**: Use search data from HBase and/or persisted results from previous jobs.

**Parameters**:

- **stats**: String 'all' or List with subset of ['avg', 'median', 'sum', 'min', 'max', 'count', 'kurtosis', 'skewness', 'stddev', 'variance']
- **perform_search**: Bool to flag whether you want to search for parameters in the job, defined in the client side script.
- **resolution**: Can be String 5m, 30m, 1d, OR any int in microseconds
- **previous_results**: Dict defining which job_id's and which result types you want to reuse in the job

**Return**: None

Cluster side script with statistics

```
import pyares as pa
ares_job = pa.init_aresjob()
ares_job.define_statistics_job(stats='all', perform_search=True, resolution='5m')
```

pyares.init_aresjob.execute_job

pyares.init_aresjob.execute_job(script, param_names, start, end, script, wait_completion, venvs, succeeded_cb, failed_cb, status_cb, check_interval)

Use only in the **client side script**.

Executes the Ares job on the cluster. Needs the script containing the functions to execute, as well as the data (params, start, end).

**Parameters**:

- **param_options**: List of strings with parameter name(s), or list of dictionaries with keys 'name' and 'get_raw'
- **start**: Timestamp with the start of the period
- **end**: Timestamp with the end of the period
- **script**: Python script (inclusion of path optional)
- **wait_completion**: Boolean for wait until job is completed. Default is False.
- **venvs**: Python Virtualenv to include in the job
- **succeeded_cb**: Function that is called once the job successfully finishes, with arguments job and status (optional)
- **failed_cb**: Function that is called if the job fails, with arguments job and status (optional)
- **status_cb**: Function that is called every check_interval seconds to get an update about job progress, with arguments job and status (optional)
- **check_interval**: Time to wait between job progress updates for status_cb. Default 10 seconds. (optional)

**Return**: String with application id

```
import pyares as pa

params = ['A', 'B']
start = 1388534400000
end = 1388620800000

# Executing a job without callbacks, just returns the job id
job_id = ares_job.execute_job(script='example_job.py', param_names=params, start=start, end=end)

# Define callbacks for different job events (success, failure, status update)
def succeeded_cb(job, status):
    print('Job successfully finished')
def failed_cb(job, status):
    print('Job failed')
def status_cb(job, status):
    print('Job status update')
    print(status)

ares_job.execute_job(script='example_job.py', param_names=params, start=start, end=end,
```
**pyares.init_areasjob.get_job_status**

Use only in the client side script.

Gets the job status for a given job id. Job id is optional and part of the class instance, which persists the application id of the last submitted job. Returns the status as a dictionary with keys Progress, State, and Final-State.

Note: Progress does not update, e.g. is stuck at 10% until completion, because the ARES cluster resource manager (YARN) is not compatible with PySpark. This might get resolved in future versions. The status is informative to know whether the job is still running, complete or has failed.

**Parameters:**

- job_id: String with complete application id, e.g. 'application_1234567890123_1234'

**Return:** Dictionary with status.

```python
import pyares as pa
ares_job = pa.init_areasjob()
params = ['A', 'B']
start = 1388534400000
end = 1388620800000

job_id = ares_job.execute_job(script='example_job.py', param_names=params, start=start, end=end)

ares_job.get_job_status()
```

**pyares.init_areasjob.get_job_logs**

Use only in the client side script.

Gets the job logs for a given job id. Job id is optional and part of the class instance, which persists the application id of the last submitted job. PyAres logs a number of events that can be reviewed with this function.

**Events:**

- “Running PyAres job.”
  - Default for every job that gets submitted correctly.
- “Search returned no data. Please consider using different parameters or a different time period.”
  - If the given parameter(s) and time period returns an empty result from HBase, this event gets logged.
- “Parameter A was not found in the database or was not found within the given time frame. Please check if this is a valid parameter name and consider using a different time frame.”
  - If the HBase search did not return empty, but one parameter could not be found, this is the event that gets logged.
- “Cannot find parameter of query %s in result given current columns %s.”
  - If the UDF dictionary function_dict passed to the define_udf_job() function has parameters in the query that are not included in the search or previous results, the UDF cannot be executed and this event gets logged.
  - E.g. if the dictionary looks like `{example1: (func1, ['A','C'])} but the search has params = ['A','B'], the log might look like cannot find parameter of query ['A','C'] in result given current columns ['A'].
- “Finished PyAres job.”
  - Default for every job that gets submitted and does not exit unexpectedly.

**Parameters:**

- job_id: String with complete application id, e.g. 'application_1234567890123_1234'

**Return:** String with all relevant log lines

```python
import pyares as pa
ares_job = pa.init_areasjob()
params = ['A', 'B']
start = 1388534400000
end = 1388620800000

job_id = ares_job.execute_job(script='example_job.py', param_names=params, start=start, end=end)

ares_job.get_job_logs()
```
Job Submission Usage Example

Executing an Ares Job requires two pieces of code: the script calling the execution and the script defining the functions. The script calling the execution of the job also declares what data will be processed. For this example the Client Side script will not change, only the Cluster Side script.

Client Side script

```python
import pyares as pa

# Define callbacks for different job events (success, failure, status update)
def succeeded_cb(job, status):
    print('Job successfully finished')

def failed_cb(job, status):
    print('Job failed')

def status_cb(job, status):
    print('Job status update')
    print(status)

params = ['sa','sx','sy','sz']
start = 1379240000000
end = 1379240000000

aj = pa.init_aresjob()
job_id = aj.execute_job(script='spark_job.py',
                        param_names=params,
                        start=start,
                        end=end,
                        succeeded_cb=succeeded_cb,
                        failed_cb=failed_cb,
                        status_cb=status_cb)
```

Cluster Side Script Example 1: Simple statistics job

```python
import pyares as pa

# We want to calculate statistics over the search results, so we don't need anything else.

ares_job = pa.init_aresjob()
ares_job.define_statistics_job(perform_search=True,
                               resolution='5m')
```

Cluster Side Script Example 2: Two UDFs

```python
import numpy as np
import pyares as pa

def calc_avg(data) -> float:
    return float(np.mean(data))

def calc_add(data) -> float:
    return float(np.add(data, 100))

aj = pa.init_aresjob()
result = aj.define_udf_job(f=[calc_add])
```

How To Define a UDF

In this section you can find more information about the way PyAres expects the UDFs to be defined.

Syntax

Important Note 1: UDFs need to be defined as element wise functions, so for every element in the column of a DataFrame, you want to apply the function. You can add the values of two columns, but you cannot aggregate over the entire column.

Important Note 2: UDFs need to be annotated with their result type. Because of the translation from Python objects to JVM objects, we need to static type our functions, e.g. def udf(x) -> int. Supported types are `int`, `float`, `str`.

Syntax of a PyAres UDF:
def <name> (arg1, arg2, ..., argn) -> return_type:
    <element wise functionality>
    return return_type(result)

Here the element wise functionality refers to any operation or sequence of operations you could do when the input arguments contain a single value and not a more complex data type like list() or dict(). And while you could create repetition structures that use these single value arguments, like iterating over a range of numbers defined by the arguments, the result also should reduce to a single return value.

Example of an allowed repetition structure:

# Arbitrary function to calculate the number of items in any interval
def count_interval_items(start, end, stepsize) -> int:
    count = np.sum([1 for number in range(start, end, stepsize)])
    return int(count)

# Alternative version
def count_interval_items(start, end, stepsize) -> int:
    count = 0
    for number in range(start, end, stepsize):
        count += 1
    return int(count)

Example of a faulty repetition structure:

# Arbitrary function that gets the numbers from a range between x and y
def range_return(x, y) -> list:
    somelist = []
    for number in range(x, y):
        somelist.append(number)
    return somelist

Using Libraries for UDFs

Using certain libraries for UDFs is not a problem, while others pose some difficulty.

Packages that pose no problem:

- numpy; this is a dependency of PyAres and is therefore part of the available packages in the Spark nodes
- Python built-ins; any default built-in library (re, math, time, etc.) that is part of the PyAres used Python version

Packages that do cause problems:

- pandas; pandas support is only available for versions of PySpark above 1.6, so not in our current implementation
- sklearn; the current implementation of PyAres does not warrant usage of sklearn, and is therefore not included in the dependencies

Usage: just import as part of the Cluster Side script, e.g.:

import numpy as np

def someudf(x) -> int:
    return int(np.add(x, 100))

Examples of UDFs

Example of a Float returning function

# Function that calculates a radian from an angle
import numpy as np

def angle_to_rad(degrees) -> float:
    radian = np.radians(degrees)
    return float(radian)

Example of an Integer returning function

# Function that checks for presence of substring in input string
# Returns a boolean as an int
import re

...
```python
def has_umbra(event) -> int:
    pattern = re.compile(r'UMBRA')
    if re.search(pattern, event):
        return (1)
    return int(0)
```

Example of a **String** returning function

```python
# Function that turns a continuous variable into a categorical one
# based on predefined bins
def cat_trafo(x) -> str:
    if x <= 10:
        return 'A'
    elif x > 10 and x <= 20:
        return 'B'
    else:
        return 'C'
```

**Result Retrieval**

The results from the Ares job can be retrieved back to the client with the result retriever.

**pyares.init_param_jobresultretriever**

```python
pyares.init_param_jobresultretriever(conf_obj=pyares_config)
```

Initializer for the constructor that automatically sets up the connection to HDFS based on the config file.

```python
import pyares as pa
result_retriever = pa.init_param_jobresultretriever()
```

**pyares.init_param_jobresultretriever.get_job_result_df**

```python
pyares.init_param_jobresultretriever.get_job_result_df(job_id, result_type)
```

Gets the result of the job from HDFS for a given job id in the form of a pandas dataframe. Depending on the definitions in the job submitted, the results can be from the search, the statistics and/or the user defined functions.

**Result Types:** (as defined in the Ares Job section of the API)

- **search**: The persisted samples from the searched parameters.
- **stat**: The result from the statistics job for a previously defined resolution.
- **udf**: The result from the user defined functions from the submission script.

**Parameters:**

- **job_id**: String with application id of the job you want to retrieve results from
- **result_type**: String either 'search', 'stat' or 'udf' for the type or result to be retrieved.

**Return**: Pandas DataFrame of specified job result

```python
import pyares as pa
result_retriever = pa.init_param_jobresultretriever()

job_id = 'application_1234567890123_0001'
result_retriever.get_job_result_df(job_id=job_id, result_type='stat')
```

**Result Retrieval Usage Example**

```python
import pyares as pa
from pyares import pyares_config

# Add custom configuration file
pyares_config.set_conf('/some/path/somefile.ini')

# Initialize the result retriever.
result_retriever = pa.init_param_jobresultretriever()

# Use job_id from previous job.
job_id = 'application_1234567890123_0001'

# Get the result data as a Pandas DF.
```
Full Usage Example

Example 1: Data Exploration Included

In this example we use the full pipeline of exploration, execution and result evaluation. Job callback functions are used for job success, failure and status update events.

Client Side script

```python
import pyares as pa
from pyares import pyares_config

# Add custom configuration file
pyares_config.set_conf('/some/path/somefile.ini')

# Initialize the needed datasources.
data_provider = pa.init_param_sampleprovider()

# Get some paras that you want to look for and of course some time period as well
params = ['A', 'B']
timestamp_start = 1387324800000
timestamp_end = 1388584759000

# Get the data from HBase and transform into Pandas DF
df = data_provider.get_parameter_data_df(params, timestamp_start, timestamp_end)
print(df.shape)

# Define callbacks for different job events (success, failure, status update)
def succeeded_cb(job, status):
    # Job succeeded, get the actual result with the result retriever
    retriever = pa.init_param_jobresultretriever().get_job_result_df
    job_id = job.get_job_id()
    search = retriever(job_id=job_id, result_type='search')
    stat = retriever(job_id=job_id, result_type='stat')
    udf = retriever(job_id=job_id, result_type='udf')
    # Do something meaningful with the results
    print(udf)

def failed_cb(job, status):
    # Job failed, get the logs to check if anything went wrong inside the job
    # that can be resolved from the client side
    print(job.get_job_logs())
    print(‘Job failed.’)

def status_cb(job, status):
    print(‘Status:’)
print(status)

# Once you have checked your data and are happy with what you will get back,
# you submit the script defined below to the cluster...
aj = pa.init_aresjob()
job_id = aj.execute_job(script=‘example_job.py’,
                        param_names=params,
                        start=start,
                        end=end,
                        succeeded_cb=succeeded_cb,
                        failed_cb=failed_cb,
                        status_cb=status_cb)
```

Cluster side script

```python
import pyares as pa

def udf1(x, y) -> float:
    return float(x-y)

def udf2(x) -> str:
    if x >= 100
        return ‘0’
    else:
        return ‘1’

function_dict = {‘example_a’: (udf1, [‘A’, ‘B’]),
                 ‘example_b’: (udf2, [‘A’])}
```
Example 2: Previous Results Only

In this example we run the option of only executing our user defined functions on results from a previous job. For this we have to know the application id of the previous job, so it is important to keep track of your previous jobs if you want to use this functionality.

Cluster Side Script

```python
import pyares as pa

def udf1(x, y) -> float:
    x = int(x)
    y = int(y)
    return float(x-y)

def udf2(x) -> str:
    if x >= 100
        return '0'
    else:
        return '1'

function_dict = {'example_c': (udf1, ['example_a','example_b']),
                 'example_d': (udf2, ['example_a'])
                 }

# We want to calculate udf1 and udf2 over the results from a previous job,
# which we know has column 'example_a' and column 'example_b'
# We need the application id from the previous job, as well as the result set that these columns appear in.
# In this case, we know the result appears in the udf result set
previous_results = {'application_1234567890123_1234': ['udf']}

ares_job = pa.init_aresjob()
ares_job.define_udf_job(function_dict=function_dict,
                        perform_search=False,  # We don't want to search for any new parameters
                        previous_results=previous_results)  # We add the dictionary with the previous job definitions
```

Client Side Script

```python
import pyares as pa

# Initialize the needed datasources.
data_provider = pa.init_param_sampleprovider()

# Define callbacks for different job events (success, failure, status update)
def succeeded_cb(job, status):
    print(job.get_job_logs())
    # In this example we also only create udf results
    retriever = pa.init_param_jobresultretriever().get_job_result_df
    udf = retriever(job_id=job.get_job_id(), result_type='udf')
    # Do something meaningful with the results
    print(udf)

def failed_cb(job, status):
    print(job.get_job_logs())
    print("Job failed.")

def status_cb(job, status):
    print(status)

# Get the paramnames from the previous job that you want to reuse
params = ['example_a', 'example_b']
start = 1387324800000
end = 1388584759000

# Submit the script defined below to the cluster...
aj = pa.init_aresjob()
job_id = aj.execute_job(script='example_job.py',
                        param_names=params,
                        start=start,
                        end=end,
                        succeeded_cb=succeeded_cb,
                        failed_cb=failed_cb,
                        status_cb=status_cb)
```

Example 3: Statistics Only
In this example we want to run a small statistics job on our parameters, because we might need the statistics to plot a graph for a report.

Cluster Side Script

```python
import pyares as pa

# We want to calculate statistics over the search results, so we don't need anything else.
# We decide on a 1h resolution

ares_job = pa.init_aresjob()
ares_job.define_statistics_job(perform_search=True,
                              resolution='1h')
```

Client Side Script

```python
import pyares as pa

# Define callbacks for different job events (success, failure, status update)
def succeeded_cb(job, status):
    print(job.get_job_logs())

    # In this example we also only create stat results, so that's what we'll be retrieving
    retriever = pa.init_param_jobresultretriever().get_job_result_df
    udf = retriever(job_id=job.get_job_id(), result_type='stat')

    # Do something meaningful with the results
    print(udf)

def failed_cb(job, status):
    print(job.get_job_logs())
    print("Job failed.")

def status_cb(job, status):
    status = job.get_job_status()
    print(status)

params = ['A', 'B']
start = 1387324800000
end = 1388584759000

# Submit the script defined below to the cluster...
aj = pa.init_aresjob()
job_id = aj.execute_job(script='example_job.py',
                        param_names=params,
                        start=start,
                        end=end,
                        succeeded_cb=succeeded_cb,
                        failed_cb=failed_cb,
                        status_cb=status_cb)
```