RADBOUD UNIVERSITY

MASTER THESIS

Improving space weather analysis using Spark Structured Streaming

Author:
Roel BOUMAN

Supervisor:
Prof. Dr. Arjen DE VRIES

A thesis submitted in fulfillment of the requirements for the degree of MSc Computing Science

in the

Data Science Group
Institute for Computing and Information Sciences

March 11, 2020

"I love deadlines. I love the whooshing noise they make as they go by."

Douglas Adams, The Salmon of Doubt

RADBOUD UNIVERSITY

Abstract

Faculty of Science
Institute for Computing and Information Sciences

MSc Computing Science

Improving space weather analysis using Spark Structured Streaming

by Roel BOUMAN

In this thesis we show how the Spark platform and the Spark Structured Streaming API can be leveraged in order to improve space weather data analysis. We build upon the existing processing pipeline in use at ASTRON by using a virtualized Spark cluster in order to perform online normalization operations instead of performing the procedure in a batch-wise manner. We show how Spark Structured Streaming can be used in order to easily incorporate new sources of data into an analysis pipeline by joining streams from different sources in an online manner. Additionally, an online power filter procedure is developed in order to save time in manual annotation of space weather data gathered by the LOFAR telescope. We finally provide extensive supplements to allow the reader to reproduce and improve upon all developed pipelines.

Acknowledgements

Met deze scriptie komt dan toch eindelijk het einde van mijn tijd als student. Ik heb het geluk gehad om erg veel te mogen leren, op academische wijze, maar ook daarbuiten.

Ik wil allereerst mijn bestuursgenoten van bestuur 13 van A.S.V. Karpe Noktem bedanken. Eva, Koos, Lenno, Max, Sophie en Willanka, we hebben een geweldig, productief, leerzaam, en bovenal slopend bestuursjaar gehad, ook al zijn er momenten geweest waarop we nauwelijks nog door dezelfde deur konden.

Tijdens mijn studie heb ik het geluk gehad met veel gezellige en leuke mensen samen te mogen werken. Tijdens mijn stages bij de chemometrie afdeling heb ik altijd veel hulp gehad aan mijn collega's die altijd bereid waren om even te praten over het willekeurige wiskundige probleem waar ik die week mee zat.

Van Arjen, Arthur, en Tom heb ik de kans gekregen om verschillende vakken te assisteren en me volop uit te leven met het verbeteren van studiemateriaal en het helpen van studenten, het was leukste en meest voldoenende werk dat ik me voor kan stellen.

Door de vele jaren heen heb ik altijd vrienden om mij heen gehad waarmee ik de nodige tabletop games en bordspellen heb kunnen spelen, alsmede daarbuiten ook de nerd uit te hangen. Het zijn veel groepen geweest waarmee we om tafel hebben gezeten, maar ik wil jullie toch allemaal bedanken. De eerste groep, waarmee we op Vossenveld met 12 man een belachelijk zelfgemaakt spel speelden elke week; de pathfinder groep, waarin we meer dan 3 jaar een letterlijk fantastisch avontuur beleefden; de dark heresy groep, waarin we tot groot ongenoegen van de spelleider ons eigen ruimteschip opbliezen; de "it's a tentacles" groep, waarmee we nog steeds, zij het met een kleine hiatus, gekke avonturen beleven in een stad vol reuzen; en tot slot de groep die ik zelf nu door onze tweede campaign heen mag begeleiden en die het al meer dan 4 jaar volhouden met mijn gestoorde fantasie.

Mijn ouders en familie moeten ook absoluut bedankt worden. Door drukke perioden heb ik jullie niet altijd zo vaak kunnen bezoeken als dat ik graag had gewild, maar jullie waren daarin altijd erg meelevend. Menig keer dat jullie bij mij langskwamen omdat ik het te druk had om bij jullie te komen hebben we het alsnog erg gezellig gehad. Dat de afwas en de helft van de andere huishoudelijke klusjes daarna meteen ook maar gedaan was is een leuk iets om aan terug te denken, maar hopelijk is het na het afstuderen niet meer nodig!

Ik wil voor de begeleiding van deze scriptie Arjen, Maaijke, en Albert-Jan bedanken. Ik vond het geweldig om de kans te krijgen om voor ASTRON onderzoek te doen. De faciliteiten in het kleine Dwingeloo zijn geweldig, en de applicatie vond ik geweldig interessant. Ik heb van jullie de kans gekregen veel nieuwe dingen te leren, zowel over space weather, als over data engineering. Arjen, ik wil jou specifiek bedanken voor de hulp met mijn dagelijkse begeleiding. Met jouw kritische vragen en opmerkingen is het je gelukt om mij altijd een stapje verder te helpen en het onderzoek de goede kant op te sturen.

Als laatste wil ik Laura bedanken. Laura, zonder jou had ik nooit alles kunnen doen wat ik gedaan heb. Je hebt jezelf meermalen opzij gezet om mij te helpen wanneer ik het nodig had, zelfs wanneer jij al zat op je bordje had. Ik hoop dat ik na mijn studie iets minder "chronisch druk" ben en dat wij nog een hele tijd alle leuke en gestoorde dingen in het leven mogen delen.

Contents

List of Figures

List of Abbreviations

ADC	**A**nalog-to-**D**igital **C**onverter
API	**A**pplication **P**rogramming **I**nterface
CEP	**CE**ntral **P**rocessing
CME	**C**oronal **M**ass **E**jection
CSV	**C**omma-**S**eparated **V**alues
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DAL	**D**ata **A**ccess **L**ibrary
ESA	**E**uropean **S**pace **A**gency
FLOPS	**FL**oating **P**oint **O**perations **P**er **S**econd
GPU	**G**raphics **P**rocessing **U**nit
HBA	**H**igh-**B**and **A**ntenna
HDF5	**H**ierarchical **D**ata **F**ormat **5**
JSON	**J**ava**S**cript **O**bject **N**otation
JVM	**J**ava **V**irtual **M**achine
KNMI	**K**oninklijk **N**ederlands **M**eteorologisch **I**nstituut, translation: Royal Netherlands Meteorological Institute
LBA	**L**ow-**B**and **A**ntenna
LOFAR	**Lo**w **F**requency **Ar**ray
ORC	**O**ptimized **R**ow **C**olumnar file format
OS	**O**perating **S**ystem
RAM	**R**andom-**A**ccess **M**emory
RDD	**R**esilient **D**istributed **D**ataset
RFI	**R**adio-**F**requency **I**nterference
SEP	**S**olar-**E**nergetic **P**articles
SKA	**S**quare-**K**ilometer **A**rray
SSA	**S**pace-**S**ituational **A**wareness
UDF	**U**ser **D**efined **F**unction
VM	**V**irtual **M**achine

Chapter 1

Introduction

1.1 Space Weather

Modern society is becoming increasingly dependent on ICT and other electronics. While this technology is becoming more advanced it is also getting increasingly vulnerable to disturbances caused by the Sun [1]. Many of our appliances, both those located on the surface of the Earth, as well as those in orbit around it, can be critically affected by the Sun in many ways. These effects can have a notable impact on the societal workings of mankind as well as the global economy [1] [2] [3].

The most notable solar events interfering with our technology and research enterprises are listed below [4]:

1. **Flares**, which are sudden increases in solar brightness. These flares occur on the surface of the Sun often close to a sunspot [5] A flare produces electromagnetic radiation in a wide spectrum of wavelengths, from radiowaves to X-rays. Flares are caused by a sudden release of magnetic energy in the Sun's corona. The X-rays radiation, as well as UV radiation, may disrupt the Earth's ionosphere and thereby interfere with radio emissions for communication, as well as radars or other technology operating in the same wavelengths. Flares often co-occur with Coronal Mass Ejections, or CMEs, the mechanism through which they are related is not yet well understood. Due to this co-occurence, flares have often been seen as the cause of disturbances actually caused by CMEs [6].

2. **Solar Energetic Particles, or SEPs**, which are particles moving at near-relativistic, i.e. close to the speed of light, are generated during solar storms and can reach the Earth in a few minutes. These particles are especially harmful to equipment and astronauts outside the Earth's magnetosphere [4], but are also dangerous on high-altitude flights on polar routes [7].

3. **Coronal Mass Ejections, or CMEs**, which are clouds of ionized gas often over 1 trillion kilograms ejected from the Sun [8]. They reach the Earth on a timescale of hours to days and cause geomagnetic storms, disrupt radio transmissions, and cause power outages [9, 10].

These solar events are colloquially known as *space weather*, and specifically refer to space surrounding the planet Earth. Currently, space weather can hardly be forecast [11, 12]. It is hard to predict even major events, as well as the effects of events occurring at the surface of the Sun. Both experimentally developed methods of forecasting and modeling from a solar physics perspective are lacking.

As forecasting space weather is hard, we can only rely on real-time observations in order to minimize economic impact of space weather. These observations are made with either ground-based or satellite-based monitoring equipment which is often dedicated specifically to space weather. A radio telescope can be used to observe solar radio bursts, which can in turn be used to forecast space weather in real-time [13]. Different types of bursts exist, all of them indicative of different solar events. Normally, 5 types of bursts, Type I to V, are recognized[1]. These solar radio bursts have durations from seconds up to several days, depending on their type, and can be observed in a frequency range of 10kHz up to 2GHz. We will review the various monitoring stations on or surrounding Earth, but will rather focus on the Dutch radio telescope LOFAR, the source that we use in this thesis.

1.2 The Low-Frequency Array, LOFAR

The LOw-Frequency ARray, more commonly known as LOFAR, is a modern radio telescope built and designed by ASTRON, the Netherlands Institute for Radio Astronomy [14]. The telescope stands out in several ways. Firstly, it does not consist of a single antenna, but as the name indicates, is made up of an array of smaller antennas. LOFAR knows two different types of antennas, a low-band antenna, or LBA, and a high-band antenna, or HBA. LOFAR consists out of several stations, each of which consists out of 96 LBA dipoles and 48 to 96 HBA tiles.

The LBA is designed to observe frequencies from 10MHz up to 90 MHz. In practice, this range of frequencies can not be properly observed, as the FM band, used for radio transmissions, interferes at frequencies higher than 80MHz, while the signal below 30MHz suffers from other sources of RFI. A photograph of the LBA antenna can be found in Figure 1.1. In this photograph it can be seen that the antenna is fixated on a conducting ground plane constisting of a metal wire frame in order to minimize vibrations in the antenna. It should be noted that the antenna itself can not be moved, but instead observes the entire sky, any directional information is extracted through digital beam-forming, which will be elaborated upon later in this Section.

The HBA can observe frequencies from 110MHz to 250MHz, although frequencies above 240MHz are generally not observed due to the presence of RFI in that range. Each HBA consists out of 16 antenna elements in a 4-by-4 grid contained in a tile. A picture of part of a HBA tile with the protective covering removed can be found in Figure 1.1. Because each tile consists out of 16 antenna elements, the signal of each tile is beam-formed in an analog manner, in contrast to the LBA, which is only beam-formed digitally. The beam-formed signals of multiple HBA tiles are further processed through digital beam-forming.

Each LOFAR station consists out of multiple HBA tiles and LBA dipoles. The signals of these antennas are first digitized and then processed in a processing station located on site. The main processing steps are amplification, digitization, filtering and beam-forming of the signal. In beam-forming, the raw signals of the antennas are combined into a directional signal by adding the signals whilst correcting for the position of each grid, and the direction in which one wants to observe. Due to the constructive and deconstructive interference of the various signals a single directional signal can be obtained. The antennas are in a fixed position but can, through digital beam-forming, observe specific parts of the sky. Multiple directions can be observed at once, as long as there is sufficient computing power to perform multiple simultaneous beam-forming steps.

At the time of writing the LOFAR telescope consists out of 51 LOFAR stations, 38 of which are located in The Netherlands, and 13 of which are located throughout

[1]For a more detailed overview of types of solar radio bursts and their properties, see the website of the Australian Bureau of Meteorology.

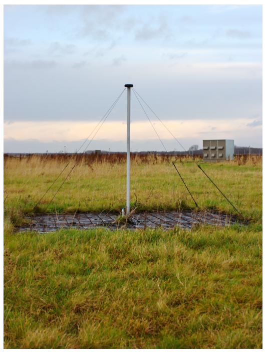
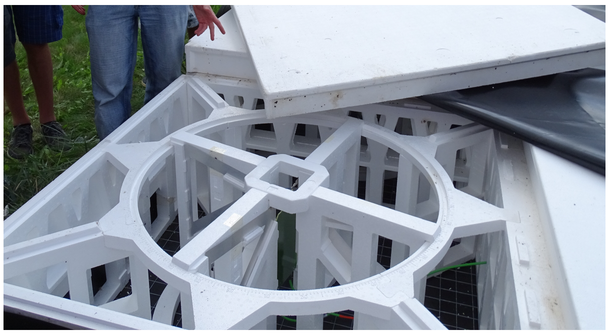

FIGURE 1.1: Photographs of the LOFAR LBA and HBA antennas. (left) A photograph of a LOFAR LBA antenna. Photograph made by A.R. Offringa distributed through the CC BY 3.0 license. (right) A photograph of part of a LOFAR HBA antenna. The protective covering is removed to show the inside. Photograph made by B. Wolba distributed through the CC BY 4.0 license.

Europe. The inclusion of these widely spread out stations allows for a larger baseline, allowing for a higher resolution observation of the sky.

The data acquired at the station level, both nationally and internationally, is transported to the city of Groningen through high speed light-paths using 10 Gigabit Ethernet. The Central Processing facility, or CEP, is located there. The CEP consists out of both an online and an offline section. The online section is made up out of custom designed GPU based setup with specifically designed correlation software called Cobalt [15]. The system consists out of 10 Dell T620 nodes each containing two Nvidia Tesla K10 GPUs. 2 out of these 10 nodes are used as spare and development nodes, while 8 are used in operation. the system relies heavily on the CUDA GPU programming platform. During online processing, the data sent in from the stations undergo several operations and transformations. While we will not elaborate all of these in detail, it should be noted that the input data are among other operations, synchronized, correlated, and integrated. These operations are costly since the high time resolution data of many stations are phaseshifted multiplied and integrated. It is the mere datarate that makes the operations costly, multiple directions do not increase the costs, since the bandwidth is limited, so more directions means less bandwidth per direction. Further offline processing is also performed, but by a separate cluster of 100 nodes assigned to both storage and computation. The total cluster has a 2 Pbyte storage capacity and a 20.6 TFLOPS peak performance. Many further offline operations exist, including the detection of calibration, image-creation, and the monitoring of data quality. We will elaborate further on the applications of operations on space weather in Section 1.3.

Due to the aforementioned ability of LOFAR to observe multiple directions at once because digital beam-forming is a digital process, the Sun can be observed continuously during the day. This allows for the gathering of large quantities of space weather relevant data, whilst still allowing LOFAR to be used for other research purposes. LOFAR research with respect to solar weather has the capability to trace CMEs from launch to the Earth, hoping to elucidate the exact effects of CMEs on the interplanetary magnetic field and thereby improving forecasting of geomagnetic storm impact on Earth. Recently, the use of LOFAR for continuous space weather observations has also been formulated as the main objective in the LOFAR4SW project for the Horizon 2020 EU framework programme [16]. In the LOFAR4SW project, LOFAR will be used for space weather monitoring next to regular astronomical observations.

In addition to its innovative capabilities LOFAR also acts as pathfinder, a form of early prototype, for the Square Kilometer Array, or SKA, project [17]. The large computing requirements of LOFAR are essential in developing processing pipelines for the SKA.

1.3 Improving LOFAR and space weather data analysis using Spark Structured Streaming

From Section 1.2 it is evident that computer science, and database management specifically, plays an essential role in the operation and development of the LOFAR telescope. Many low-level parts of the data gathering and processing pipeline have been thoroughly optimized through a rigorous iterative process [18, 19, 20]. Nonetheless, many improvements can still be made on a higher level, specifically in many of the data analysis pipelines after initial beam-forming and processing has been done.

Indeed, it has been recognized that collaboration between domain experts, such as astronomers, and computer scientists is essential for scaling data-focused research [21]. In their technical report Gray et al. describe how natural science could profit from working closely with computer scientists. Their findings indicate that research questions can be answered faster due to the implementation of proper database and data processing systems instead of relying on older methods based on the batch-wise analysis and transportation of the data. This collaboration will become increasingly important as the development of the SKA progresses. The computational requirements of the SKA are extensive [17]. Preliminary investigations on SKA requirements have shown that maintainability, reusability, availability, performance, reliability, scalability, and reproducibility are all essential attributes for the SKA [22]. These essential attributes are present in the Spark platform, which could also be leveraged for the SKA after the development for the SKA pathfinder LOFAR. We will outline these features of Spark in more detail in Section 2.1.

In collaboration with ASTRON we have identified several applications within LOFAR research which could benefit from recent advances in data engineering. We will specifically focus on those applications which we feel can be improved upon using the recently developed Spark Structured Streaming API [23]. The Structured Streaming API allows a data scientist to formulate operations and queries for an online streaming application in a declarative manner. This allows for an easy transition from current declarative and batch based procedures to online data analysis. In addition the Apache Spark framework offers numerous benefits for large scale distributed computing. We will further elaborate on Apache Spark and Spark Structured Streaming in chapter 2. In this Section we will instead only introduce the possible applications of Spark Structured Streaming within LOFAR and space weather analysis in general.

1.3.1 Space weather research with LOFAR

LOFAR has in recent years been used to minitor space weather. This project aims to enable the use of solar, solar wind and ionospheric observations with LOFAR in order to monitor, predict, and model solar events impacting Earth. LOFAR can in addition be used to measure in the solar, heliospheric, and ionospheric domains simultaneously. Through these observations the spectral data that is gathered can be used in order to detect various solar events. One of the major advantages of the use of

LOFAR for space weather monitoring is that it can potentially be done continuously due to the digital beam-forming that is employed using these type of antennas. As long as beam-forming computational capacity is available, relevant spectral data can be gathered.

Currently data analysis on the LOFAR project includes real-time preprocessing and visualization of the spectral data that is being gathered. This is currently done in a batch-wise manner. The pipeline operates on raw beam-formed data after the CEP is done processing. An example plot of beam-formed data can be found in figure 1.2. The pipeline is mainly written in the Python programming language [24]. A simple exploratory analysis of the beam-formed data that is used in this pipeline can be found in Appendix A.1. This appendix shows in what manner the data is saved to disk after beam-forming, and shows what information is saved.

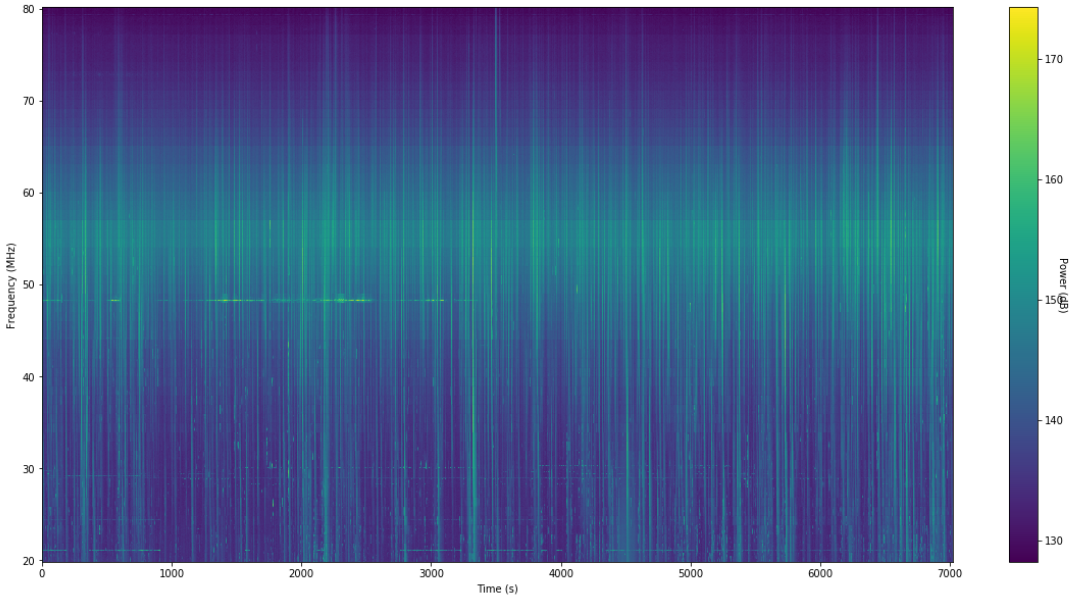

FIGURE 1.2: Plot of beam-formed Stokes shift data. The data has been log-transformed in order to reduce the dynamic range. so as to better visualize extreme values. Note that the data is subsetted so as to only include 1 in every 300 samples in order to circumvent memory limitations. The Y-axis indicates the wavelength of the measurement, while the X-axis indicates the time of acquisition.

The current data analysis pipeline has several weaknesses and limitations that could be alleviated or solved by adapting it to use the Spark Structured Streaming API.

The pipeline is first of all operated in a batch-wise manner. Structured Streaming can be used to make many operations, including the training of classifiers or performing pre-processing operations, be performed continuously. This allows for a better distribution of computational workload, as well as more frequent intermediate results. The better distribution of workload can also allow for higher resolution spectral analysis, as less sub-sampling is needed.

Secondly, the project has several distinct goals. By translating these to separate streaming pipelines they can be implemented, and optimized separately. Separate pipelines could for example be used to have classification pipelines using different window sizes for the detection of different event types, and the real-time visualization.

The last major advantage of using Structured Streaming in the analysis of beam-formed data is the ability to do streaming SQL join operations. ASTRON researchers have expressed interest to incorporate different data sources, such as live weather data from the station's location, into their classification and calibration models. Online join operations can in this manner speed up batch processing and ensure that as soon as data arrives, processing can begin. Incorporating multiple data streams is easy to express using the Spark Structured Streaming API, which can help in developing more accurate classification and calibration methodology.

Spark Structured Streaming additionally offers other computation advantages, such as monitoring, failure and straggler resistance, easy rescaling, and code updates without having to fully recompute intermediates. We will outline these features in Sections 2.1 and 2.2.

1.3.2 Other space weather applications

The European Space Agency, or ESA, has been identifying systems for Space Situational Awareness, or SSA, using machine learning. In this programme they identify currently used tools to monitor, predict, and understand space weather, as well as early-warning systems. As a part of this project several expertises of various groups in Europe have been identified who are working on SSA. This collective offers several services regarding space weather, including, but not limited to, flight crew radiation exposure prediction, monitoring and forecasting of ionospheric disturbances, and quality assesments of ionospheric corrections. Many of their key objectives, such as the forecasting of radiation storms, risk estimates of micro-particle impacts, and the effects of space weather on sensitive flight electronics remain unfulfilled. Although we will not cover any of these applications in details, the faster prototyping of possible models and the increased speed of data analysis in general offered by the application of Spark Structured Streaming in machine learning for astronomy might aid in fulfilling these goals faster.

1.3.3 The scope of this research

In this thesis we will show how data analysis for the LOFAR solar data project can be improved and made more maintainable by making use of the Apache Spark project and the Spark Structured Streaming API in particular. We will show how the major computational workload of current LOFAR analysis for space weather can be done on a Spark cluster in an online manner. We show how beam-formed data can be normalized, power filtered, and joined with other data in a streaming environment. While we present many concepts specifically for the LOFAR solar data application, the research can be easily generalized to similar applications within ASTRON, or as a prototype for future ESA or SKA research. In order to provide reproducible and extensible results, all experiments and developed methodology are documented and supplied in the Appendices A, B, and C.

Chapter 2

Theory

2.1 Apache Spark

In this research we will leverage the well-known Apache Spark engine. Apache Spark is a computing engine for clusters which focuses on automatic parallelization of both data and processing and is fault tolerant. Spark has initially been developed by Zaharia et al. at UC Berkeley [25] [26], but has since been donated to the Apache foundation. In contrast to many other systems used in big data and relational database applications it aims to provide a general framework for big data processing [26].

Apache Spark shares a programming model with the MapReduce framework [27], but distributes data in a unique manner. To share data across nodes, Spark relies on a shared data abstraction called the Resilient Distributed Dataset, or RDD. A RDD is fault-tolerant, and abstracts a data object so it can be distributed over a cluster. As the data is distributed over a cluster, it can be operated on in parallel. RDDs can be transformed in several ways. A user might define a set of transformation on an initial RDD in order to acquire a result. By employing lazy evaluation, these transformations are combined in an operator graph, and subsequently combined into an evaluation plan. Spark can optimize such an evaluation plan, combining transformations, so as to require fewer passes over the entire dataset. It can for example combine several map and filter operations. Spark will also minimize the amount of shuffling and moving of data across nodes that happens across an evaluation plan. These optimizations and the intelligent sharing of data allow for large speed increases over other computational frameworks [28], as well as enable the generality of the Spark framework. RDDs are inherently fault tolerant [29], and in case of node failure, RDDs are (partially) recomputed by using the constructed graph of an evaluation plan of a query. Further optimizations of Spark include workload balancing performed by the Spark master in a cluster, which also accounts for stragglers, for which resistance is built in. Although Spark performs comparably to other engines, its generality comes at a cost. In comparative research, it is has been concluded that it requires more memory as it introduces more overhead [30].

More recently the Spark framework has been extended to use the dataframe abstraction. Dataframes are a way of storing tabular data, and have become increasingly popular in data science, as is evidenced by the popular Pandas library for Python [31] [32] and the dataframe abstraction in the R language [33] [34]. Internally, dataframes are represented in Spark as RDDs, allowing for implicit parallelization of data and fault-tolerance.

Spark also has its own SQL engine, known as Spark SQL, which makes use of the tabular representation of Spark dataframes [35]. Spark SQL is heavily optimized and uses the same graph-based optimization procedure as used for RDD transformation. Using these optimizations, Spark can perform comparably to specialized SQL engines [26].

Spark additionally is supplied with its own machine learning library, MLlib. [36]. Many commonly used algorithms are implemented, including Alternating Least Squares, Latent Dirichlet Allocation, and k-means clustering. Although we do not make use of the MLlib library in this research, the existence of machine learning support in Spark is essential for future applications within LOFAR data analysis research.

Spark natively runs on the Java Virtual Machine, or JVM, but can be easily used from multiple languages common in data analysis, most notably Scala, Java, Python, and R. Spark is generally deployed on top of a distributed file system such as S3, Cassandra or HDFS. Spark can be used using a variety of resource managers, such as Mesos, Yarn, Kubernetes, or using its own standalone resource manager.

2.2 Spark Structured Streaming

One of the main features that distinguishes the Apache Spark framework from others for our use-case is the Spark Structured Streaming API [23]. Spark Structured Streaming is a declarative streaming API included in Apache Spark. Its main distinguishing feature, other than operating natively on a Spark cluster, is the way in which streaming queries can be expressed. Instead of having to chain operations, a user can use simple SQL-like queries or similar declarative operations to transform a stream input. Structured Streaming leverages the SQL engine and optimizations that are present in the Spark framework. In Structured Streaming, queries are incrementalized by treating a stream as a series of microbatches, and combining them based on the query plan. Many other parts of the Spark stack can be used in conjunction with Spark Structured Streaming. Nearly all Spark SQL functionality can also be applied on Structured Streaming, and large parts of MLlib can be applied on streaming data, as long as the models are trained on static datasets. Depending on the a few basic guarantees from the data source, Structured Streaming is fault tolerant and it can be restarted from any point during operation, allowing for rapid prototyping on large query plans.

In Structured Streaming a stream can comprise one of several data sources. Supported streams are filestreams, such as a folder where CSV, Parquet, Orc, or JSON files are being written continuously; a Kafka source, where an Apache Kafka stream is being read. Additional socket or rate sources exist, but these are only implemented for testing purposes, as they do not provide fault-tolerance.

Many operations in Structured Streaming rely on event time based windows. For example join operations between two streams can be performed in Structured Streaming. When joining two streams, the entire tables have to be in memory, as the last received element of one table could match with the first received element of the other table. In order to circumvent this problem, every stream can be watermarked with a window based on a timestamp column in the streaming input dataframe. A window size can now be set in order to limit the parts of the tables that should be kept in memory, i.e. the parts of the tables that are allowed to be joined. One can for example define the window for the first table to be 5 minutes, and the window for the second table to be 1 hour. In this case, only the last 5 minutes of the first table that are received are allowed to be joined with the last hour of the second table. A late tolerance can additionally be set, so that data is allowed to be received with a delay. These windowed operations can also extend to SQL-like operations, as any aggregations in Spark SQL can only be performed on a stream that is grouped by a certain window frame.

Results from a Structured Streaming query can be output to several sinks. The most common writers are a file sink, which can write to a file in the ORC, JSON, Parquet, or CSV format; a Kafka sink, which stores the output to a Kafka topic; or a custom writer which handles custom logic for either each row, or for each microbatch.

Recently Spark has introduced a continuous processing mode [37]. Using the continuous processing mode the computational workload is distributed more evenly across time, as the computations are no longer performed only on the arrival of a new microbatch. Continuous processing is currently still in an experimental stage, and is only supported for Kafka input streams and Kafka sinks. In addition, only a select part of the operations available in microbatch operating mode are available in continuous processing as of the time of writing. Continuous processing mode currently only supports projection and selection operations, as well as a subset of SQL functionality, specifically aggregations are not yet supported.

In this research Structured Streaming was chosen as the streaming framework for several reasons. Firstly it outperformed Apache Flink and Kafka Streams on the Yahoo! Streaming benchmark [23, 38]. Secondly, the simple declarative API allows for easy formulation of query plans, especially when compared to Google Dataflow and Flink's DataStream API, which require much more in-depth knowledge of the operators, system, and manual incrementalization in order to build a manually build a query plan. Lastly, as Structured Streaming is integrated with the rest of the Spark stack, developing streaming applications which employ a variety of operations, such as machine learning and SQL queries, is easy without having to resort to using multiple frameworks and APIs. Due to its simple declarative and expressive API, pipelines can easily be updated and maintained.

Chapter 3

Experimental

3.1 The simulated Spark cluster

In our experiments we heavily depend on the Spark framework. This computing framework normally runs on a physically separate cluster, where the data and code is being sent to. Or alternatively, where the data itself is stored, but only the code is being sent to.

In this research we had no access to a physical cluster running Spark. To nonetheless demonstrate the applicability of Spark on the ASTRON use-case, we have set up a simulation of a Spark cluster by employing Docker[1].

Docker is virtualization software that virtualizes an Operating System, or OS. In contrast to a Virtual Machine, or VM, docker containers share resources with an underlying OS, sometimes called piggybacking, allowing for containers to be lightweight in RAM, CPU, and bandwidth usage[39]. Multiple Docker containers can be run on a single machine with relatively little being required in terms of additional hardware.

Our experimental cluster consists of 3 Docker containers linked through a Docker network. We differentiate between the containers by linking a single Spark master with 2 Spark workers. Each of these containers, which we will additionally refer to as nodes, has been assigned a memory of 2GB, so as to fit in the memory of the local machine[2]. Each of the nodes runs the 3.10 version of the lightweight Alpine Linux distribution[3]. On top of Alpine, Spark version 2.4.1, Hadoop version 2.7, and Python version 3.6.9 are installed. We also install the Python libraries Cython [40] [41], Numpy [42] [43], and Pandas [31] [32] as they are specifically needed for later experiments.

Instructions on how to set up a virtual Spark cluster using Docker can be found in Appendix B. Instructions are included on setting up a basic cluster in section B.1, running basic Spark examples in section B.2, and setting up the more intricate cluster, which was used in the presented experiments, in section B.3.

3.2 Incorporating multiple data sources by joining streams

With a working Spark cluster, albeit a virtual one, we can demonstrate the usefulness of Spark Structured Streaming for LOFAR experiments. In section 1.3.1 we have established that researchers at ASTRON want to incorporate diferent data sources other than the LOFAR measurements into future analysis pipelines. One such an example is the incorporation of real-time weather data. In the following experiment

[1]`https://www.docker.com/`

[2]The local machine is an older system with 8GB of DDR3 RAM, and an AMD A8-5500 APU processor.

[3]`https://alpinelinux.org/`

we will show how different data sources can be incorporated into data analysis using Spark Structured Streaming. To describe this experiment in more detail, we will first elaborate briefly upon how real-time streaming was emulated when only historical data was available.

3.2.1 Emulated streaming of data to the virtual Spark cluster

Data gathered by LOFAR is processed by the CEP in various ways as outlined in section 1.2. The major processing step that is used for the analysis of space weather using LOFAR is the beam-forming of the raw measurements. This beam-forming is performed by the CEP, and the result is written to a hdf5 file in ICD3 format. Effectively the result is spread over 2 files, a `.raw` file containing the binary data, which is being written during the experiment, immediately after the CEP has beam-formed the raw data, and a `.h5` file containing metadata pertaining to the experiment, which is written before the experiment starts. More detailed information on the file structure can be found in Appendix A.

In current real-time analysis performed by ASTRON, the `.raw` file is treated as a binary buffer-like stream. The current processing of these streams by ASTRON is further elaborated upon in section 3.3 and Appendix section C.1

In order to stream the beam-formed data to the cluster, we have developed a timed procedure which will stream a certain number of beam-formed measurements to the cluster. This is done simply by reading the `.raw` file as a buffer, thus reading measurements from it. The read measurements are streamed to the cluster by writing them to `.csv` files on the cluster. During this procedure, the beam-formed measurements, consisting only of intensities at certain wavelengths at a certain time, are also annotated by the time at which they were measured, this information is additionally added to the `.csv` file.

The second stream we initiate is a stream of data gathered from the Royal Netherlands Meteorological Institute, the KNMI. The KNMI provides an API through which weather information from measuring stations throughout the Netherlands can be gathered. A Python wrapper for this API called KNMY[4] exists, which was used to fetch the data directly into the Python environment. The developed streaming pipeline uses the timestamps of the beam-formed measurements to automatically fetch the KNMI data at that time for the measuring station Hoogeveen, which is located closest to the LOFAR center. The gathered KNMI data is streamed to the cluster by writing a `.csv` file in cluster storage.

The procedure through which data is streamed to the cluster is covered in more detail, including code listings, in Appendix section B.4.2.

3.2.2 Reading and processing the streams using the Spark cluster

Combining the two datastreams, the beam-formed LOFAR data and the KNMI weatherdata, in Spark Structured Streaming is simple to express. After both the streams have been initialized, an SQL-like join operation can be performed. Initialization of the streams can be done by defining the schema, so the number and types of variables that are read from the stream, as well a providing information relevant to reading the `.csv` files that make up the stream. Sample code for stream initialization can be found below.

[4]`https://pypi.org/project/knmy/`

```
beamformedFieldTypes = [StructField("V"+str(i), DoubleType(), False)
 ↪  for i in range(0,960)]
beamformedFieldTypes.append(StructField("beamformedtimestamp",
 ↪  TimestampType(), False))
beamformedFieldTypes.append(StructField("hourly_beamformedtimestamp"⌋
 ↪  , TimestampType(),
 ↪  False))
beamformedSchema = StructType(beamformedFieldTypes)
beamformedDF = spark \
  .readStream \
  .option("sep", ",") \
  .option("header", "true") \
  .schema(beamformedSchema) \
  .csv("/opt/spark-data/beamformed")

weatherFieldTypes = [StructField("V"+str(i), DoubleType(), True) for
 ↪  i in range(0,22)]
weatherFieldTypes.append(StructField("weathertimestamp",
 ↪  TimestampType(), False))
weatherSchema = StructType(weatherFieldTypes)
weatherDF = spark \
  .readStream \
  .option("sep", ",") \
  .option("header", "true") \
  .schema(weatherSchema) \
  .csv("/opt/spark-data/weather")
```

Joins between two streams are normally unbounded, meaning that without constraints all data must be kept in memory in order to check if certain rows should be joined. In order to solve this issue we must define how late data is allowed to arrive at the cluster before it is no longer to be considered. In addition, any join operations must also include constraints based on the timestamps of both tables. In this case, we have set arbitrary threshold for demonstration purposes, allowing the beam-formed data to arrive 1 hour late, and the KNMI data to arrive 2 hours late. Sample code for setting the watermarks based on columns in the table, as well as joining with time constraints, can be found below. It should be noted that the weather data is only delivered every hour. Due to the hourly delivery, the beam-formed timestamp has been rounded in the equals part of the join condition, e.g. `hourly_beamformedtimestamp = weathertimestamp`.

```
watermarkedBeamformedDF =
 ↪  beamformedDF.withWatermark("beamformedtimestamp", "1 hours")
watermarkedWeatherDF = weatherDF.withWatermark("weathertimestamp",
 ↪  "2 hours")

joinedDF = watermarkedBeamformedDF.join(
  watermarkedWeatherDF,
  expr("""
    hourly_beamformedtimestamp = weathertimestamp AND
    beamformedtimestamp >= weathertimestamp AND
    beamformedtimestamp <= weathertimestamp + interval 1 hour
    """),
  "leftOuter")
```

This joined table is still a streaming result, and it can now be used in any online data analysis pipeline. Incorporating additional sources is easily possible using similar expressions. This allows for easy extension of data analysis pipelines. In these experiments we have not developed a pipeline which uses the joined table, but rather only verify that the joining operation has worked by performing a simple aggregation which select the maximum beam-formed timestamp for a given combination of weather timestamps and beam-formed timestamps, for which the code can be found below.

```
maxTimeStamp = joinedDF.groupby("beamformedtimestamp",
↪  "weathertimestamp").agg(max_("beamformedtimestamp"))
```

An example output of the result of the joining procedure can be found in Section 4.1.

This joining procedure could also be used to prepare newly acquired data for later analysis when multiple sources are needed. For such a purpose, it might be advantageous to write the resulting joined table to disk.

The procedure through which the datastreams are processed on the cluster is covered in more detail, including code listings, in Appendix section B.4.4.

3.3 Online processing of beam-formed LOFAR data

Currently at ASTRON, beamformed data from solar observations is being processed in an online fashion using a Python script. This script reads the observations from a `.raw` file connected to a layer of hdf5 information in the form of a `.h5` layer.

The script reads the `.raw` file as a buffer, and processes each read, with a maximum of 10000 samples per read. The numerical data is normally subsampled 1 in every 25 measurements over the time axis. Every one of these batches, with a maximum size of 400-by-the number of frequencies measured, is then scaled by dividing it by the median of this batch, a procedure called normalization by ASTRON. Each of these scaled batches is appended to a larger matrix-like data structure which is visualized for each iteration of this procedure. In addition to the visualization of the scaled data, the most recently calculated median is also shown. This differs from the procedure used in the current processing pipeline, where the median is not calculated per batch, but rather includes all measurements available.

In our experiments we have adapted the ASTRON analysis to instead stream the data to our Spark cluster. On the Spark cluster the data is scaled and the results are written to disk. The written results are subsequently plotted outside of the Spark cluster using a Python script, in a procedure similar to the original ASTRON procedure.

In the following Section 3.3.1 we elaborate on the implementation of the online scaling procedure. We also expand upon the current ASTRON pipeline by applying an online power filter, in order to detect interesting events, in section 3.3.2. More details on the original ASTRON procedure can be found in Appendix section C.1.

3.3.1 Online median scaling of beam-formed data

In this section, we show how Spark Structured Streaming can be leveraged in order to replace the batch-wise median calculation with an online calculation using the

Greenwald-Khanna algorithm [44]. Using Structured Streaming, we can express the median calculation as a windowed groupBy operation, followed by an aggregation calculation to determine the approximate 50% percentile.

We can express an operation calculating the median of all variables by expressing an approximate percentile aggregation over each beam-formed variable of a streaming dataframe where we group over a time-based window. The central code calculating this streaming median can be found below.

```
exprs = [expr('percentile_approx('+x+', 0.5)').alias('med_'+x) for x
↪  in beamformedDF.columns[:960]]

medianDF = beamformedDF.withWatermark("beamformedTimestamp", "5
↪  seconds") \
  .groupBy(
    window("beamformedTimestamp", "5 seconds", "5 seconds") \
).agg(*exprs)
```

This specifies that Spark will compute the median over every non-overlapping window of 5 seconds, with a 5 second tolerance for data that arrives late. The median can be outputted to any mode supported by Spark. In Appendix section C.3.1, where we cover this procedure in more detail, we chose to output a subset of variable medians to the console, but practical purposes might include writing the results to disk or to a Kafka stream.

The major disadvantage of calculating the median using SQL-like operations, as we have shown here, is that very few operations can be performed on streaming aggregated dataframes. This prohibits us from using the calculated median to normalize the data. Ideally we would do this by joining the calculated median dataframe with the dataframe with the beam-formed data, and subsequently dividing the beam-formed columns by their respective window-based medians. As these operations, specifically the join operations, are not (yet) supported in Spark as of version 2.4.X, or in the upcoming 3.0 release, we will have to resort to performing the scaling operations in a different manner.

We have identified two ways of circumventing these limitations in Spark, while still leveraging the other advantages Spark offer. The first way is to write the calculated medians to a Kafka stream, then re-import the Kafka stream as a new datasource, and then perform the joining and division operations. Streaming to Kafka, and re-importing it does however introduce a substantial amount of overhead and latency. The second way, which has been implemented in this research, is by using a recently introduced feature in Spark, the `forEachBatch` writer. Using this sink, arbitrary Python code can be run on each Spark micro-batch. We can now easily use the Pandas library in order to calculate the median, and divide the data by it. The advantage of this approach is that there is little additional overhead and the median and scaled data can be written separately, but it allows for less control over the window size, which is now equal to the micro-batch size and thus controlled by Spark. The code snippet covering the micro-batch based procedure can be found below. Additionally, the procedure is covered in more detail in Appendix section C.3.2.

```
def foreach_write(df, epoch_id):
  dataDF = df.select(variableNames).toPandas()
  bfTimestamp = df.select("beamformedTimestamp").toPandas()
  bfSecondsAfterMeasurement =
   ↪  df.select("secondAfterMeasurement").toPandas()
```

```
writeColumns = variableNames + ["beamformedTimestamp"]

median = dataDF.median()

scaledDF = dataDF.divide(median)

scaledDF["secondAfterMeasurement"] = bfSecondsAfterMeasurement
scaledDF["beamformedTimestamp"] = bfTimestamp
scaledDF = scaledDF.sort_values("secondAfterMeasurement")

scaledDF.to_csv("/opt/spark-results/median_scaled_data/scaled_data
 ↪  " + str(epoch_id) + ".csv", header=True, index=False,
 ↪  columns=writeColumns)
median.to_frame().T.to_csv("/opt/spark-results/medians/median" +
 ↪  str(epoch_id) + ".csv", header=True, index=False)
```

The above function is applied to each micro-batch. From each dataframe we select the numerical variables, divide them by their median, and write both the median and the scaled dataframe to disk.

The written results can now be easily read and plotted in real-time by a different Python script. This procedure is described in more detail in Appendix section C.3.2.

3.3.2 Applying an online power filter on beam-formed data

As the application of LOFAR for space weather analysis is still in an early phase, very little has been done in terms of machine learning to detect solar events. Although annotations from other sources could be transferred over based on the event-time, this would lead to a low resolution on the time axis as well as the chance of missed events, as LOFAR is much more sensitive than other equipment. Consequently, LOFAR data will have to be annotated manually in order to maximize its potential.

As manual annotation is a costly procedure, measures to reduce the time spent annotating are often employed. One such measure is the application of a simple power filter. A power filter is a simple tool. One takes the sum of squares for each measured time point, so the sum of all the squared variables, and compares that to a threshold. Every time point with a sum of squares higher than the threshold is said to be a region of interest. The regions of interest can then be annotated by domain experts, saving substantially in annotation time.

We have implemented a simple power filter, which is executed online using Spark Structured Streaming. As this research was not carried out by domain experts, the threshold has not yet been set, and instead only the sum of squares calculation is performed offline. It should however be noted that the thresholding procedure could trivially be implemented in the Spark Structured Streaming framework. We have additionally developed simple visual tools in order to select an appropriate threshold based on domain knowledge. As the thresholding procedure is not carried out online, and the sum of squares is written to disk, the thresholding procedure can also be optimized more easily for later experiments.

The implementation of a sum of squares based power filter is easy, and a code snippet containing the essential calculation expressed in the Spark Structured Streaming API can be found below.

```
variableNames = ["V"+str(i) for i in range(0,960)]

sumOfSquaresUdf = udf(lambda arr: sum(pow(a,2) for a in arr),
 ↪  DoubleType())

beamformedDF = beamformedDF.withColumn('sumOfSquares',
 ↪  sumOfSquaresUdf(array(variableNames)))
```

Chapter 4

Results

In this section we will elaborate upon the results of the experimental pipelines and operations we have developed. This research presents many proofs-of-concept, but has few quantitative results. As such, this section will consist mostly out of example output of the developed methods, showing that they work, whilst leaving quantitative comparisons to later research which might implement them.

4.1 Joining of streams

Using Spark Structured Streaming data from different streaming sources can be combined easily and in an online manner. In section 3.2 we describe in detail how the procedure works. The joined data that is the product of the developed pipeline can be further processed in several ways. In a production environment the results will likely be used for further analysis in a pipeline, or they can be written to disk for analysis on a specified subset of the data. By performing the operation online we can save in the memory requirements of the pipeline, as the two to-be-joined tables need not be loaded into memory fully. In addition, as the procedure is performed online, the fully joined data is available for further processing at an earlier stage.

To illustrate that an online joining operation works, we have joined weather data from the KNMI station Hoogeveen to the beam-formed data based on timestamps. To show an intermediate output we have performed a groupby operation on the beamformed and weather timestamps followed by an aggregation where we select the maximum beamformed timestamp. This aggregation is largely redundant, but allows for a simple console output of two identifying columns of the resulting joined dataframe, leaving the redundant aggregation as a third column. Sample output for the first few microbatches can be found below:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:00|2019-04-13 12:00:00|    2019-04-13 12:00:00|
|2019-04-13 12:00:01|2019-04-13 12:00:00|    2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|    2019-04-13 11:59:59|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 1
-------------------------------------------
```

```
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 3
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 4
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 5
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 6
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
|2019-04-13 12:00:17|2019-04-13 12:00:00|     2019-04-13 12:00:17|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 7
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:19|2019-04-13 12:00:00|     2019-04-13 12:00:19|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
|2019-04-13 12:00:18|2019-04-13 12:00:00|     2019-04-13 12:00:18|
+-------------------+-------------------+-----------------------+
only showing top 20 rows
```

In this example we have chosen the stream to start just before the measurement time reaches the full hour 12:00:00. This time was chosen so it can be illustrated that the beam-formed data is correctly joined with the latest available KNMI weather data, which updated only once per hour, rather than multiple times each second. We can also see that the joined table grows as more microbatches are processed. It should be noted that the output provided by Spark is not sorted, as the original order is not preserved throughout processing. It should therefore be ensured the data is always supplied with a timestamp, so the original ordering can be reconstructed when needed.

This experiment therefore shows the online joining capability of two streams of Spark Structured Streaming, and additionally shows that data from different sources can easily be used in conjunction with the already available beam-formed data acquired by LOFAR.

4.2 Online processing of beam-formed LOFAR data

In this research we have investigated two different applications of Spark Structured Streaming for LOFAR space weather analysis. We will first show the results of our online median scaling procedure. Secondly, we will show results of an online power filter that is run in addition to the median scaling procedure.

4.2.1 Online median scaling of beam-formed data

Using Spark Structured Streaming, the normalization procedure ASTRON uses to scale its beam-formed data can be performed in an online manner. Based on the procedure initially developed by ASTRON, we have developed an alternative procedure producing similar results. Note that we do not elaborate further upon the online median calculation procedure leveraging the Spark SQL engine outlined in the first part of section 3.3.1, as this was not used in the end-to-end pipeline.

In Figure 4.1 we show sample output of the procedure at 4 different points in time. The figure is extended whenever new data has been successfully processed and written to disk. Using the developed script, the data that is processed by the Spark Structured Streaming pipeline outlined in section 3.3.1 is plotted in near-real-time. The script is provided with a command-line interface, allowing the user to set the wait-time between updates, and to optionally show the median that was used for normalization. In Figure 4.1 the wait-time is set to 5 seconds, and the median is shown below each dynamic spectrum.

In Figure 4.1 we can observe several distinguishing features of space weather. The yellow color indicates a high intensity. Based on this high intensity we can observe several events occurring over time plotted time frame. Events can clearly be observed occurring every few seconds, indicated by the near-vertical yellow structures. We can also observe a somewhat constant baseline around the 50-65 MHz region, which

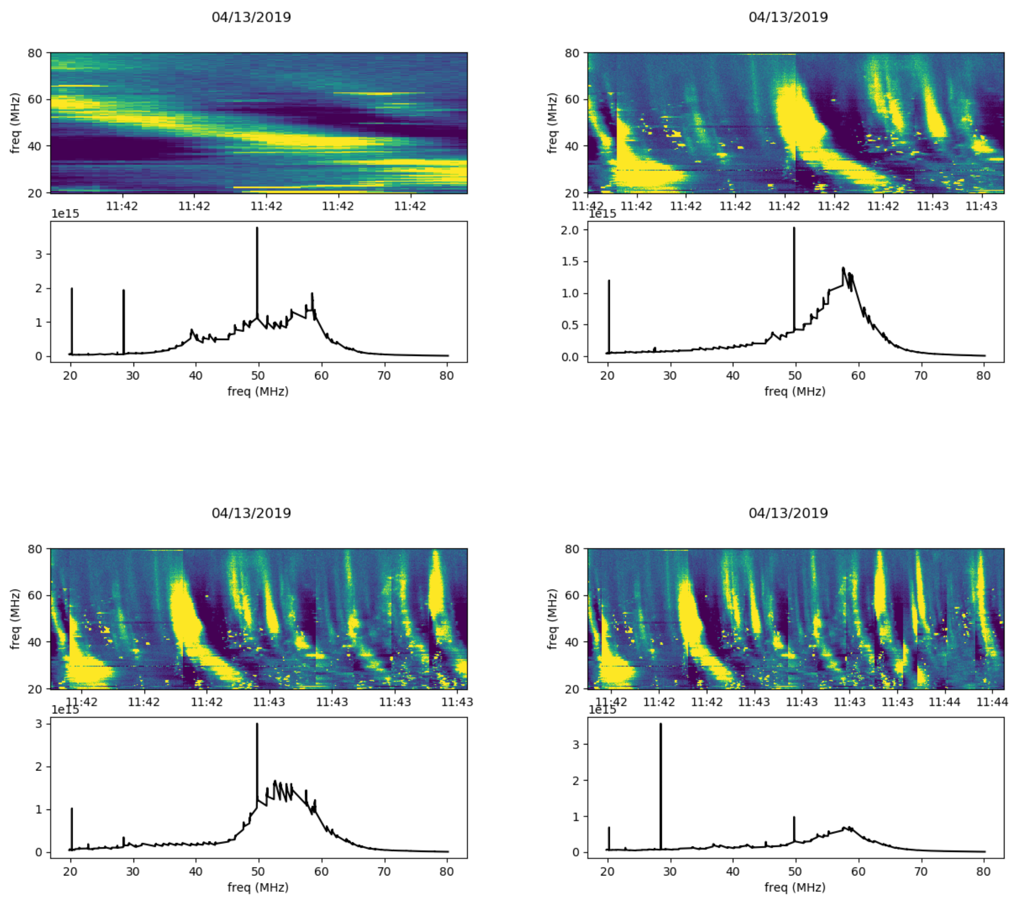

FIGURE 4.1: Intermediate results from the online normalization procedure. 4 intermediate results are shown. The upper part of each subfigure shows the normalized beam-formed, or dynamic, spectrum, and the lower part shows the most recently calculated median, i.e. the one used to scale the most recent micro-batch. In the dynamic spectrum dark blue indicates a low intensity, whilst yellow indicates a high intensity.

reflects the antenna response, that is strongly peaked at its resonance requency around 57 MHz. These short burst, combined with the underlying continuous background signal, might indicate a Type 3 Solar Radio burst storm.

We can also observe, which is especially clear in the upper-right subplot, the cut-off between the windows over which the beam-formed data is normalized. The scaling procedure inherently operates on windows, which produces these artifacts near the window borders. We can also observe several RFI peaks around 20 and 50 MHz, which might be indicative of amateur radio use, or satelite or airplane communication.

The latency with the current setup, including possible overhead from virtualization as well as spaced writing and reading operations, is approximately 30 seconds from the first write to the visualization of the plot. This can be sped up in several ways, as outlined in section 5.

4.2.2 Applying an online power filter on beam-formed LOFAR data

In addition to the median scaling procedure we have also developed a simple online power filter to be used as an initial screening method for areas of interest. Using such a screening procedure, areas of interest can be detected and manually labeled. Due to this initial screening step, large parts of the spectrum no longer require manual annotation.

We have developed a simple procedure which calculates a sum of squares based power filter using the Spark Structured Streaming API, and subsequently plots this against a user-set threshold. No thresholding is performed inside the pipeline, as it is advantageous to first determine a sensible value for the threshold. Plots of median-scaled beam-formed data with the corresponding power calculator at various stages of the real-time pipeline can be found in Figure 4.2.

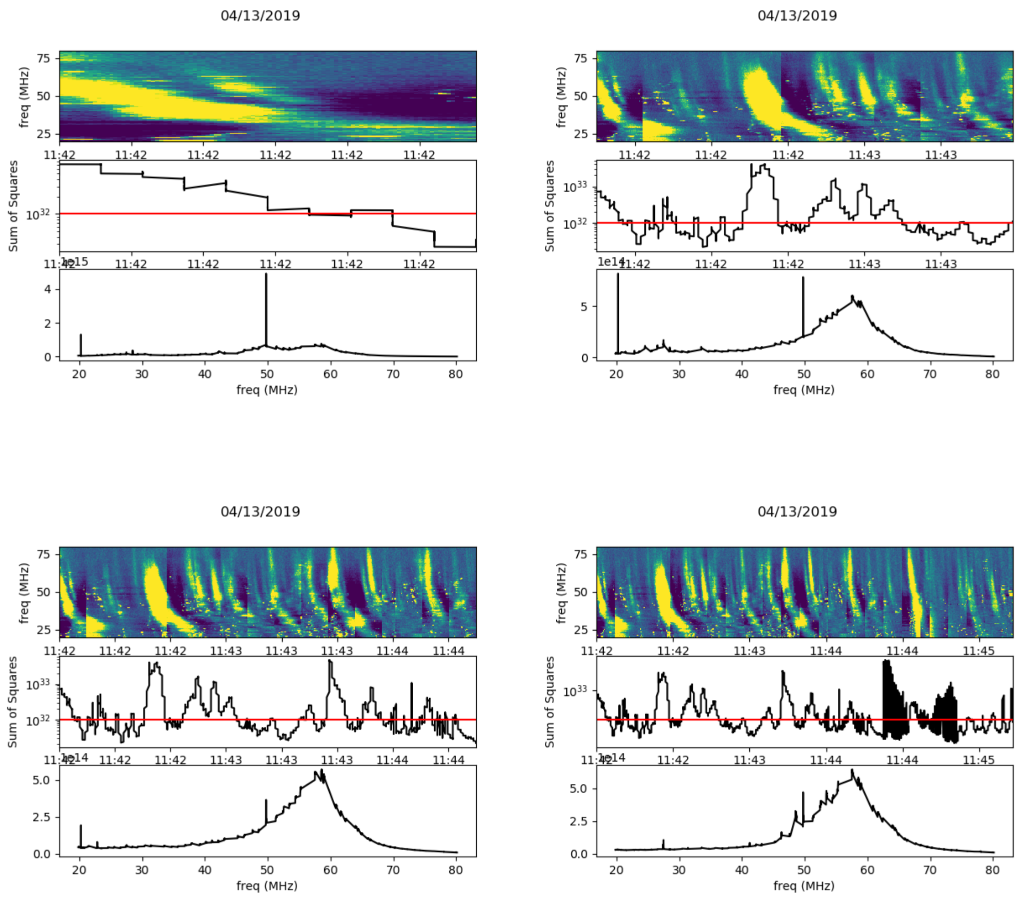

FIGURE 4.2: Intermediate results from the online normalization and power filter calculation procedure. 4 intermediate results are shown. The upper part of each subfigure shows the normalized beam-formed, or dynamic spectrum, the middle part shows the power filter as well as an adjustable threshold, and the lower part shows the most recently calculated median, i.e. the one used to scale the most recent micro-batch. The power filter threshold is set to 1e32 in this example. In the dynamic spectrum dark blue indicates a low intensity, whilst yellow indicates a high intensity.

In Figure 4.2 we can see the power filter being applied to real-time data. We can clearly see how the power filter with the chosen threshold of 1e32 can filter out what an observer can filter out by eye. The various events, indicates by the near-vertical yellow structures, correspond with the power filter above the red line. We can see that there are some deviations, and that the power-filter does not work in every case, as is to be expected for such a simple procedure. The most notable example can be seen in the the lower-right part of the figure. Between 11:44 and 11:45 large black structures can be observed in the power-filter part of the plot. Although no events can be visually observed in the beam-formed spectrum, the value of the power filter is above the threshold. This is likely caused by narrow frequency bands with very high intensities, likely caused by RFI. Although these can be seen as outliers, the current power filter does not account for these and will register false positives in such cases. Nonetheless, this simple procedure still allows for a rough identification of the areas of interest.

The power filtering procedure introduces barely any additional latency on top of that present in the procedure where the data is only scaled by its median. The latency is therefore still approximately 30 seconds from the first write to the visualization of the plot.

Chapter 5

Discussion

Although the presented research shows how Spark Structured Streaming can be leveraged in order to process and analyze data acquired by LOFAR, many things can be improved as this research progresses and moves towards being used in a production environment.

5.1 Limitations of Spark

It should first be noted that the Spark Structured Streaming API is relatively new, and although it is by far the most extensive of its kind, many features are not yet implemented natively, sometimes limiting the applicability of the API on problems in the natural sciences.

The most major shortcoming of Spark Structured Streaming with respect to the presented research is the lack of stream-stream join operations after an aggregation has been performed on one stream. This is caused by the internal workings of Spark, which does not allow states, as watermarks are more formally called, to have multiple instances across the JVM of a Spark executor, thus leading to global watermarks. As both the original, watermarked, data and the aggregate require a state, the aggregate effectively uses the state of the original data. As the aggregate does have its own state, or watermark, it is not available for further stateful operations such as joins. This problem can be solved internally in Spark, but has so far been postponed due to other required changes[1]. Two major workarounds exist, one is leveraging the ForEachBatch writer, and the second is by writing the aggregated stream, and importing it again. The first option was employed in this research, but introduces additional overhead due to the use of native Python procedures, which are less optimized than expressing them using the Spark API. The second option also introduces noticeable overhead, as the stream has to be rerouted, for example through Apache Kafka, which additionally requires a Kafka setup.

Another shortcoming is that the Spark Structured Streaming API does not provide a native method for the calculations of medians, although many other SQL engines provide such functionality. In addition, more efficient methods for the calculation of quantiles exist, such as the Frugal method [45]. Implementation of an optimized median calculation procedure, as well as the implementation of more efficient quantile approximation methods would improve the existing pipeline.

5.2 Further optimization of the developed procedure

The presented methodology shows, as a proof of concept, what can be done using the Spark Structured Streaming API. Nonetheless, the presented procedures can still

[1] `https://github.com/apache/spark/pull/23576`

be optimized. In this section we will highlight some of the major parts which can be improved in future research.

The presented methodology is mostly executed on a virtual Spark cluster deployed on several developed Docker images. These Docker images have been constructed and adapted as needed throughout this research. These images are based on templates by Big Data Europe for use in their analysis pipelines and are extended by installing specific software needed for our experiments. As we have added an additional layer on top of their Spark master and Spark worker images, a layer which is identical, many installation instructions are performed twice, instead of only once for a base image. This results in longer build times when the image is first built. Additionally, the images are not as lightweight as they could be due to specific software being installed which is not used in the presented procedures.

Currently, new beam-formed data is read from the `.raw` file using a Python procedure, and then streamed as `.csv` files to the virtual cluster. This introduces quite some I/O overhead. Ideally, a new reader would be written which can directly handle the growing `.raw` file as a stream input. With the upcoming Spark 3.0 release, binary file formats will be supported as data source. As of the time of written it is unclear whether it will also be able to be used as a data source for structured streaming. If this option is not supported it would be preferable to implement such a method specifically for this use-case, as it removes one of the major sources over overhead.

Currently, reading and writing is done using the CSV file format. The CSV file format is not very efficient in terms of storage and loading properties. Although the CSV stream used to import the data might be replaced by a binary stream with the advent of spark 3.0, the results will still need to be written. A custom writing procedure could be made in order to write to a customized binary format using the ForEachBatch writer. Another option would be to adopt the Parquet file format [46], which is substantially more optimized than using CSV files. The Parquet file format could also be used to replace the initial procedure of writing to the cluster before a binary file stream is implemented.

In the presented methodology we make use of the PySpark API for Python, rather than using the Spark API for Scala or Java, which natively run on the JVM. Using the PySpark API is slower for nearly every usecase, as it introduces additional overhead, and allows for less internal optimization by the Spark scheduler. Python was chosen nonetheless, as current data analysis by ASTRON is nearly completely performed in Python, allowing for an easier adoption of new tools. As the PySpark API is very popular among Spark developers, many improvements are being made, making it faster with each new Spark version, making this choice less impactful as development of Spark progresses.

We currently make use of a ForEachBatch procedure in order to normalize the data by scaling using a calculated median. In this procedure the internal dataframe structure used by Spark has to be converted into a Pandas dataframe for use with the untranslated Python code. This conversion introduces a substantial amount of overhead, making processing slower and computationally more expensive. Recently, the Apache Arrow project[2] has been developing a cross-language platform for in-memory data. The project can be used in Python using PyArrow, and is natively supported within Spark. Using this acceleration less conversion would be needed, substantially reducing the computational requirements of the developed pipeline. In addition, we use UDFs for the calculation of the power filter. UDFs are currently much slower when used in PySpark than in the JVM based language, even when

[2]`https://arrow.apache.org/`

vectorized. With the upcoming release of Spark 3.0, UDFs can be accelerated by using the Apache Arrow engine, leading to even greater speed increases when leveraging PyArrow.

In the current normalization pipeline the ForEachBatch procedure is currently triggered with every new `.csv` file that is written, this is not ideal when the batch size per file is too small or too large for meaningful medians to be calculated. A trigger time can be instated when the batch size is too small, but this trigger time depends on machine time rather than the timestamp on the data.

Spark currently employs a microbatch processing mode. Using this type of processing each microbatch is processed as soon as a trigger for it is activated. This means that the Spark cluster effectively has a short amount of downtime between the processing of a microbatch and the trigger for the processing of the next microbatch. A new processing mode for Spark Structured Streaming is currently in development, which no longer uses microbatch processing, but rather uses a true streaming approach by allowing continuous processing. Unfortunately, at the time of writing, this mode is not yet fully developed, and only supports Kafka stream sources. As Kafka stream sources are likely to be employed in production, further experimentation with continuous processing mode is desirable.

Currently we submit jobs to our cluster using the Spark standalone cluster manager. As we only submit a single job simultaneously, the standalone cluster manager fits our needs. When multiple jobs need to be submitted, a more advanced cluster manager such as Apache Mesos[3] or Apache Hadoop YARN[4] is preferable. These managers allows for a more detailed control of the resources of the cluster, and their division over various processes. Cluster managers such as Mesos or YARN additionally offer more tools for monitoring of the application and cluster workload during operation.

Currently, the end-to-end latency of the normalization and power filtering procedures is high. Using many of the aforementioned improvements it can potentially be improving to be as fast or faster than the native Python procedure employed at ASTRON while still allowing the strengths of the Apache Spark platform to be leveraged.

5.3 Additional improvements

There are several improvements which can be made to the developed procedures outside of speed, efficiency, and memory usage.

The plotting procedures can be easily improved upon by making them interactive. Using Python libraries such as Plotly[5] and its module Dash[6] interactive visualizations can be made. These interactive visualizations would for example allow for the selection of a certain median used for scaling to be shown based on selection in the plot.

The developed power filter procedure is very sensitive to outliers. When narrow band frequencies have a high intensity, for example due to RFI, the threshold for detection can be easily exceeded. A simple, yet more robust filter, can be developed easily by first normalizing the data before calculating the sum of squares. This makes the filtering method less sensitive to outliers, but can lead to values above

[3]`http://mesos.apache.org/`

[4]`https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

[5]`https://plot.ly/`

[6]`https://plot.ly/dash/`

the threshold in periods of solar inactivity, as all data is normalized with a local median. Another improvement to limit the influence of RFI is to divide the power of a subband by the differential power, where the differential power is the mean of the two neighbouring subbands.

Chapter 6

Conclusion

In this thesis we have shown how LOFAR data analysis, specifically LOFAR space weather analysis, can be carried out in an online fashion by making use of Spark Structured Streaming.

We have shown that the expressive Spark Structured Streaming API can be used to easily merge beam-formed data, that results from LOFAR measurements and CEP processing, with other data. In our application we have performed an online join operation by joining the beam-formed data with weather data gathered by the KNMI at the weather station in Hoogeveen, located near the main LOFAR antenna hub. Although the weather data is limited in detail, and it is updated only hourly, it demonstrates the type of operations that can be easily performed using Spark. Online join operations allow for easy incorporation of new data sources, allowing LOFAR data analysis to be extended and maintained more easily as new methods of analysis are added. In addition, joining with new data sources could help in labelling LOFAR data by cross-referencing with data from other space weather observation.

We have adapted ASTRON's current LOFAR space weather analysis pipeline in order to leverage the capabilities of Spark Structured Streaming. Originally, the beam-formed data is read from a binary file buffer and processed in a batch-wise manner as more binary data is added. We have converted the binary file buffer to a CSV file stream connected to a Spark cluster, process it by dividing it by its median, and write it through a CSV stream back to disk. The developed methodology acts as a proof of concept, showing that Spark can indeed be used to perform such operations, online, in a streaming setup. Many things can still be improved in the developed methodology. The current streaming setup introduces significant overhead by not making good use of the binary file buffer, which can't be natively imported as stream into Spark. Although a streaming median over a moving window can be calculated in Spark Structured Streaming, we must leverage the microbatch engine and a custom writer in order to scale the data. This is less efficient than performing continuous calculations and scaling, which is not yet possible using Spark.

We have developed a power filter method which can be applied online using Spark Structured Streaming. Using the power filter method beam-formed data can be screened for areas of interest. This screening can help in future annotation of the data for automatic classification of solar events, as current data is unlabeled and annotation is time-consuming. The developed filter is sensitive to outliers, for example by high RFI, an alternative filter based not on the raw beam-formed data, but rather on the normalized data is suggested, but is sensitive to false positives in periods of low solar activity.

Appendix A

Exploratory analysis of LOFAR data

A.1 Exploring LOFAR beamformed data

The data from LOFAR all adheres to an internal standard, specified in the "Low Frequency Radio Astronomy and the LOFAR Observatory" book [47]. A detailed guide on the beam-formed data format can be found on the public LOFARwiki[1]. The data is saved as a HDF5 file, consisting out of a `.h5` and a `.raw` file. These contain the header information and the raw data respectively.

We can do some initial exploratory analysis on the data and its structure. We will do this mostly based on the tutorial[2] provided by Cees Bassa. In addition to the lecture slides code examples on how to analyse LOFAR data are provided by Cees Bassa through GitHub[3]. Note that normally, one would analyze the data from LOFAR with DAL, the Data Access Library[4]. As we only require access to the numerical information stored inside the file, we'll rely on manual exploration of the HDF5 file. First we can start by importing the required packages and loading the data.

```
import h5py
import numpy as np
import matplotlib.pyplot as plt

filename = 'L701913_SAP000_B000_S0_P000_bf.h5'

h5 = h5py.File(filename, "r")
```

Every HDF5 file consists out of two kinds of objects, datasets and groups. The datasets are array-like, and groups are structured much like folders in any regular filesystem. A group can be accessed in Python using syntax very similar to that which we use for dictionaries. In this case we'll show the group folder-like structure.

```
def print_name(name):
    print(name)

h5.visit(print_name)
```

Which will yield:

[1]https://www.astron.nl/lofarwiki/lib/exe/fetch.php?media=public:documents:lofar-usg-icd-003.pdf
[2]`https://www.astron.nl/lofarschool2018/Documents/Thursday/bassa.pdf`
[3]`https://github.com/cbassa/lofar_bf_tutorials`
[4]`https://www.astron.nl/lofarwiki/doku.php?id=public:user_software:dal`

```
SUB_ARRAY_POINTING_000
SUB_ARRAY_POINTING_000/BEAM_000
SUB_ARRAY_POINTING_000/BEAM_000/COORDINATES
SUB_ARRAY_POINTING_000/BEAM_000/COORDINATES/COORDINATE_0
SUB_ARRAY_POINTING_000/BEAM_000/COORDINATES/COORDINATE_1
SUB_ARRAY_POINTING_000/BEAM_000/PROCESS_HISTORY
SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0
SUB_ARRAY_POINTING_000/PROCESS_HISTORY
SYS_LOG
```

We can also read the metadata of this experiment, which is contained in the upper group.

```
group = h5["/"]
keys = sorted(["%s"%item for item in sorted(list(group.attrs))])
for key in keys:
    print(key + " = " + str(group.attrs[key]))
```

Which yields will yield most of the relevant experiment related metadata, some of which we will use later:

```
ANTENNA_SET = b'LBA_OUTER'
BANDWIDTH = 23.4375
BANDWIDTH_UNIT = b'MHz'
BF_FORMAT = b'TAB'
BF_VERSION = b'Cobalt/OutputProc 3.2_6 r41591 using DAL 2.5.0 and HDF5
 ↪  1.8.12'
CLOCK_FREQUENCY = 200.0
CLOCK_FREQUENCY_UNIT = b'MHz'
CREATE_OFFLINE_ONLINE = b'Online'
DOC_NAME = b'ICD 3: Beam-Formed Data'
DOC_VERSION = b'2.5.0'
FILEDATE = b'2019-04-13T11:41:02.0'
FILENAME = b'L701913_SAP000_B000_S0_P000_bf.h5'
FILETYPE = b'bf'
FILTER_SELECTION = b'LBA_10_90'
GROUPTYPE = b'Root'
NOF_SUB_ARRAY_POINTINGS = 1
NOTES = b''
OBSERVATION_END_MJD = 58586.56874522193
OBSERVATION_END_UTC = b'2019-04-13T13:38:59.587174416Z'
OBSERVATION_FREQUENCY_CENTER = 49.993896484375
OBSERVATION_FREQUENCY_MAX = 80.169677734375
OBSERVATION_FREQUENCY_MIN = 19.818115234375
OBSERVATION_FREQUENCY_UNIT = b'MHz'
OBSERVATION_ID = b'701913'
OBSERVATION_NOF_BITS_PER_SAMPLE = 8
OBSERVATION_NOF_STATIONS = 36
OBSERVATION_NOF_SUB_ARRAY_POINTINGS = 2
OBSERVATION_START_MJD = 58586.4875
OBSERVATION_START_UTC = b'2019-04-13T11:42:00.000000000Z'
```

```
OBSERVATION_STATIONS_LIST = [b'CS001LBA' b'CS002LBA' b'CS003LBA'
 ↪  b'CS004LBA' b'CS005LBA' b'CS006LBA'
 b'CS007LBA' b'CS011LBA' b'CS013LBA' b'CS017LBA' b'CS021LBA'
  ↪  b'CS024LBA'
 b'CS026LBA' b'CS028LBA' b'CS030LBA' b'CS031LBA' b'CS032LBA'
  ↪  b'CS101LBA'
 b'CS103LBA' b'CS201LBA' b'CS301LBA' b'CS302LBA' b'CS401LBA'
  ↪  b'CS501LBA'
 b'RS106LBA' b'RS205LBA' b'RS208LBA' b'RS305LBA' b'RS306LBA'
  ↪  b'RS307LBA'
 b'RS310LBA' b'RS406LBA' b'RS407LBA' b'RS409LBA' b'RS503LBA'
  ↪  b'RS508LBA']
PROJECT_CONTACT = b'Fallows, Dr Richard'
PROJECT_CO_I = b'Caterina Tiburzi'
PROJECT_ID = b'LT10_002'
PROJECT_PI = b'Mann, apl. Prof. Dr. Gottfried'
PROJECT_TITLE = b'Advancing Space Weather Science with LOFAR and the
 ↪  Parker Solar Probe'
SYSTEM_VERSION = b'3.2_6'
TARGETS = [b'Sun' b'TauA']
TELESCOPE = b'LOFAR'
TOTAL_INTEGRATION_TIME = 7019.587174400001
TOTAL_INTEGRATION_TIME_UNIT = b's'
```

We can now for example see over what time the experiment was conducted, when the data was gathered, and which stations were used to measure.

We will now take a look a the numerical data itself, the Stokes I shift data. We'll again start by listing the group attributes.

```
group = h5["/SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0"]
keys = sorted(["%s"%item for item in sorted(list(group.attrs))])
for key in keys:
    print(key + " = " + str(group.attrs[key]))
```

Which will yield:

```
DATATYPE = b'float'
GROUPTYPE = b'bfData'
NOF_CHANNELS = [16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
 ↪  16 16 16 16 16 16
 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
  ↪  16
 16 16 16 16 16 16 16 16 16 16 16 16]
NOF_SAMPLES = 669440
NOF_SUBBANDS = 60
STOKES_COMPONENT = b'I'
```

We can for example see how many subbands are measured, and how many samples, so spectra, are measured. As the total matrix is quite large, being 60*16*669440*32 = 20.57Gbits when fully loaded, we'll only plot parts of the Stokes I shift matrix by only plotting 1 in every 300 samples in order to circumvent memory limitations:

```
stokes = h5["/SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0"]
data = 10.0*np.log10(stokes[1::300,:])

freq = h5["/SUB_ARRAY_POINTING_000/BEAM_000/COORDINATES/COORDINATE_1"]
 ↪  .attrs["AXIS_VALUES_WORLD"]*1e-6
tsamp = h5["/SUB_ARRAY_POINTING_000/BEAM_000/COORDINATES/COORDINATE_0"
 ↪  ].attrs["INCREMENT"]
t = tsamp*np.arange(stokes.shape[0])

vmin = np.median(data)-2.0*np.std(data)
vmax = np.median(data)+6.0*np.std(data)

plt.figure(figsize=(20, 10))
plt.imshow(data.T, aspect='auto', vmin=vmin, vmax=vmax,
 ↪  origin='lower', extent=[t[0], t[-1], freq[0], freq[-1]])
plt.xlabel("Time (s)")
plt.ylabel("Frequency (MHz)")
plt.colorbar().set_label('Power (dB)', rotation=270)
```

Note that we've used some of the metadata in order to have sensible axis values, and that we've converted to a log scale to better visualize the shift even though extreme values are present. The figure can be found below in Figure A.1.

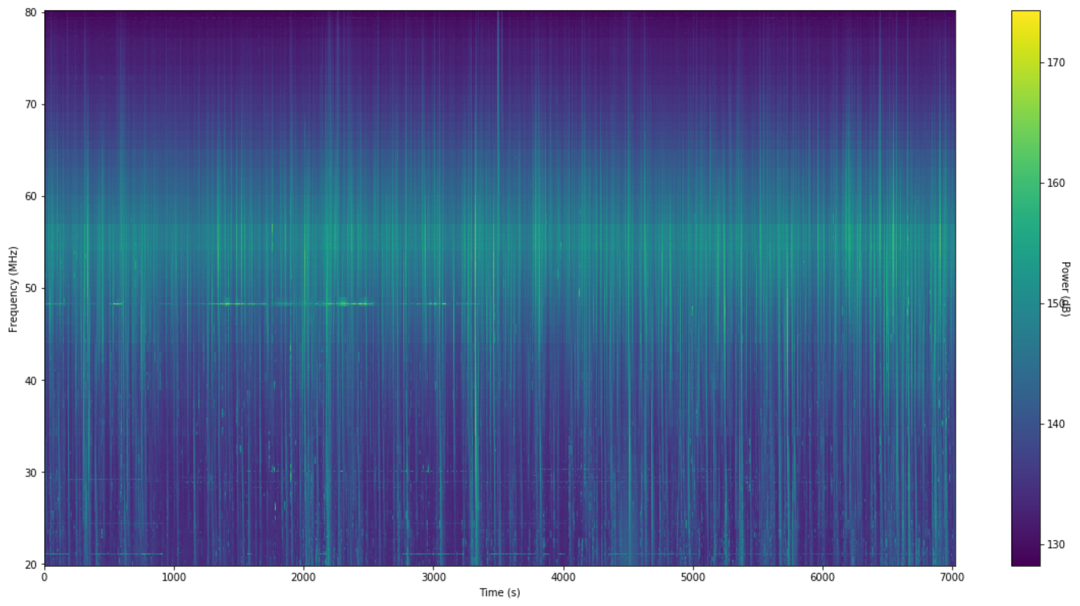

FIGURE A.1: Plot of beam-formed Stokes I shift data. The data has been log-transformed in order to reduce the dynamic range. so as to better visualize extreme values. Note that the data is subsetted so as to only include 1 in every 300 samples in order to circumvent memory limitations. The Y-axis indicates the wavelength of the measurement, while the X-axis indicates the time of acquisition.

With a visualization of the data we can now look further into the practical application of streaming for solar weather analysis.

Appendix B

Setting up the Spark Structured Streaming pipeline prototype

B.1 Setting up the basic cluster

We will first create a simple virtual cluster of a Spark master and two Spark workers. As they will be operating in unison, we need to define a network encompassing these containers. We will build on the templates of Big Data Europe, as they are preconfigured to work together, saving some time in the setup.

```
docker network create spark-net

docker create --name spark-master -h spark-master --network spark-net
 ↪  -p 8080:8080 -p 7077:7077 -e ENABLE_INIT_DAEMON=false -m=2g
 ↪  bde2020/spark-master:2.4.1-hadoop2.7
docker create --name spark-worker-1 --network spark-net -p 8081:8081
 ↪  --link spark-master:spark-master -e ENABLE_INIT_DAEMON=false -m=2g
 ↪  bde2020/spark-worker:2.4.1-hadoop2.7
docker create --name spark-worker-2 --network spark-net -p 8082:8081
 ↪  --link spark-master:spark-master -e ENABLE_INIT_DAEMON=false -m=2g
 ↪  bde2020/spark-worker:2.4.1-hadoop2.7
```

Note that we initialize the containers with limited memory, so as to not run into memory limitations in a development setting. The Docker default would normally set the maximum memory of each container to the maximum available in the system, which does not work when you are actively working with multiple containers. We will then finish setting up with starting the individual containers:

```
docker start spark-master
docker start spark-worker-1
docker start spark-worker-2
```

B.2 Testing basic cluster functionality

We can now test PySpark by submitting an example job to estimate pi from inside the container:

```
docker exec -it spark-master /bin/bash
```

Follow by the following command from within the container:

```
./spark/bin/spark-submit --master spark://spark-master:7077
 ↪  examples/src/main/python/pi.py 100
```

We should see that we can find a reasonable estimate of pi. Now in order to test a streaming example we will initialize a fourth container which streams data to the master through netcat. This should be done in a separate console, so not on a bash terminal inside the container.

```
docker run -it --rm --name nc --network spark-net appropriate/nc -lk
 ↪  9999
```

Then we can listen on port 9999, we can illustrate this by running the wordcount example of docker using the following code:

```
./spark/bin/spark-submit --master spark://spark-master:7077
 ↪  examples/src/main/python/streaming/network_wordcount.py nc 9999
```

Where the last two arguments indicate the name of the container, and the respective port where we'll be listening. Now if we can enter lines through the netcat container simply by typing some lines of text. The streaming wordcount will then display the word counts coming in every second in the container where the script was submitted. In spark standalone mode, a lot of information is being shown that is not needed for monitoring of a stream, we can turn this off by making a log4j properties file from a template and editing the logging configurations.

```
cp spark/conf/log4j.properties.template conf/log4j.properties
```

```
vi spark/conf/log4j.properties
```

And then we change

```
log4j.rootCategory=INFO, console
```

to

```
log4j.rootCategory=ERROR, console
```

Now when we run the example again we will find that the output is much more legible.

We can extend easily extend this framework to spark structured streaming, which should work out of the box with another example:

```
./spark/bin/spark-submit --master spark://spark-master:7077 examples/s⌋
 ↪  rc/main/python/sql/streaming/structured_network_wordcount.py nc
 ↪  9999
```

Again, we can stream data to our listening cluster through our netcat container. A running wordcount will be updated nicely. In addition to running these applications, we can also monitor the spark process through an UI by navigating to `localhost:8080`. Here we can see the running and completed applications, and get more insight into the load of the workers. Note that this only works when we provide the `--master spark://spark-master:7077` argument. Otherwise Spark will run in local mode, without using the cluster.

B.3 Setting up a more advanced cluster

The cluster set up until now works for reproducing the PySpark examples provided with the Spark distribution, but it lacks a few key properties we will need for a more practical emulation of a real world example. We will for example need to emulate a distributed file system. We will accomplish this by using Docker Compose to more easily set up, and shut down the cluster. We do this by making a custom `docker-compose.yml` file, roughly based on work from Marco Villarreal[1]. The contents of this file are given below:

```
version: '2'
services:
  spark-master:
    build: spark-master/.
    image: spark-master
    mem_limit: 2G
    container_name: spark-master
    ports:
      - "8080:8080"
      - "7077:7077"
    networks:
      spark-network:
        ipv4_address: 10.5.0.2
    volumes:
       - /mnt/spark-apps:/opt/spark-apps
       - /mnt/spark-data/beamformed:/opt/spark-data/beamformed
       - /mnt/spark-data/weather:/opt/spark-data/weather
       - /mnt/spark-data/metadata:/opt/spark-data/metadata
       - /mnt/spark-results:/opt/spark-results
       - /mnt/spark-results/median_scaled_data:/opt/spark-results/medi ⌋
        ↪  an_scaled_data
       - /mnt/spark-results/medians:/opt/spark-results/medians
       - /mnt/spark-results/power_filter:/opt/spark-results/power_filt ⌋
        ↪  er
  spark-worker-1:
    build: spark-worker/.
    image: spark-worker
    mem_limit: 2G
    container_name: spark-worker-1
    depends_on:
      - spark-master
    ports:
      - "8081:8081"
    networks:
      spark-network:
        ipv4_address: 10.5.0.3
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
    volumes:
       - /mnt/spark-apps:/opt/spark-apps
       - /mnt/spark-data/beamformed:/opt/spark-data/beamformed
       - /mnt/spark-data/weather:/opt/spark-data/weather
       - /mnt/spark-data/metadata:/opt/spark-data/metadata
       - /mnt/spark-results:/opt/spark-results
```

[1]https://medium.com/@marcovillarreal_40011/creating-a-spark-standalone-cluster-with-docker-and-docker-com

```
       - /mnt/spark-results/median_scaled_data:/opt/spark-results/medi ↲
        ↪  an_scaled_data
       - /mnt/spark-results/medians:/opt/spark-results/medians
       - /mnt/spark-results/power_filter:/opt/spark-results/power_filt ↲
        ↪  er
  spark-worker-2:
    build: spark-worker/.
    image: spark-worker
    mem_limit: 2G
    container_name: spark-worker-2
    depends_on:
      - spark-master
    ports:
      - "8082:8082"
    networks:
      spark-network:
        ipv4_address: 10.5.0.4
    environment:
      - "SPARK_MASTER=spark://spark-master:7077"
    volumes:
       - /mnt/spark-apps:/opt/spark-apps
       - /mnt/spark-data/beamformed:/opt/spark-data/beamformed
       - /mnt/spark-data/weather:/opt/spark-data/weather
       - /mnt/spark-data/metadata:/opt/spark-data/metadata
       - /mnt/spark-results:/opt/spark-results
       - /mnt/spark-results/median_scaled_data:/opt/spark-results/medi ↲
        ↪  an_scaled_data
       - /mnt/spark-results/medians:/opt/spark-results/medians
       - /mnt/spark-results/power_filter:/opt/spark-results/power_filt ↲
        ↪  er
networks:
  spark-network:
    driver: bridge
    ipam:
     driver: default
     config:
       - subnet: 10.5.0.0/16
```

This will set up our cluster with a few minor differences from the basic setup. The most notable change is introduced by the `volumes:` command. We can in this way emulate a distributed filesystem by sharing files across the mounted folders. The specified folders will be used in later experiments. In addition we've now assigned static IP adresses for easy monitoring. Finally, we now build the master from a custom Dockerfile, ensuring that the modified `log4j.properties` file is copied to the master immediately on initialization of the cluster. The Dockerfile is located in the `spark-master` subfolder, and has the following contents:

```
FROM bde2020/spark-master:2.4.1-hadoop2.7
COPY log4j.properties /spark/conf/
COPY spark-env.sh /spark/conf/
```

```
RUN apk add build-base python3-dev
RUN pip3 install cython
RUN pip3 install numpy
RUN pip3 install pandas
```

In addition to this, some environment variables are set, and specific software and packages are installed for later experiments in Appendices B and C. These dockerfiles are not thoroughly optimized, as is elaborated upon in section 5.

To more easily reproduce these experiments, the Docker Compose and build files, as well as configuration files, are distributed through GitHub[2], and can be easily cloned.

We can now initiate the cluster with the following commands from the folder in which our `docker-compose.yml` file is located:

```
docker-compose build
docker-compose up -d
```

We can check that this has indeed initialized all 3 containers by issuing the familiar `docker ps -a` command.

When we want to shut down and remove all containers we can also use Docker Compose, and issue:

```
docker-compose down
```

On this cluster we can also run all examples as indicated in section B.2. Due to the nature of Docker Compose, the name of the network to which the `nc` container is assigned will depend on the folder name from which the project is composed. We will need to find the right network bridge to connect the container to, which is most easily done with the following command:

```
docker network ls
```

In this case, we end up needing to issue the following command to initialize the netcat container and connect it to the network:

```
docker run -it --rm --name nc --network
 ↪  dockerfilessparkcluster_spark-network appropriate/nc -lk 9999
```

With this more advanced cluster we can now more truthfully emulate a streaming use-case.

B.4 Streaming beamformed data

Our specific use-case will however not handle such word streams, but will mostly concern numerical data. Structured Streaming supports sockets only for testing, as this input does not provide end-to-end fault-tolerance. The only two supported data input sources are Kafka and File sources. We can use the latter in our case, processing data which is added constantly added to a directory accesible to the Spark master.

Not all data types are supported to be read by Spark. Of the supported types, the CSV type is the most useful.

[2]https://github.com/RoelBouman/dockerfilesSparkCluster

B.4.1 Converting hdf5 to csv, and transferring it to the cluster

To write to CSV, we must first read the beamformed data supplied by ASTRON. In order to do this, we must first load the data. We can do this in a manner similar to that used in Appendix A. We will start by outlining the basic procedure, before moving on towards a timed simulation in which csv data is written periodically. This basic procedure can be reproduced by running the `readh5_writecsv_test.py` script, which can be found on GitHub[3].

```
import h5py
import numpy as np
import tarfile
import docker
import time
from io import BytesIO

filename = 'L701913_SAP000_B000_S0_P000_bf.h5'

h5 = h5py.File(filename, "r")
```

With the data loaded as h5, we can create a tarstream of part of the data in order to write it to the shared storage of our cluster[4].

```
stokes = h5["/SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0"]

data_part = stokes[:10,:]

np.savetxt("tempdata/foo.csv", data_part, delimiter=",")

data_part_string = open("tempdata/foo.csv", "r").read()

tarstream = BytesIO()
tar = tarfile.TarFile(fileobj=tarstream, mode='w')
file_data = data_part_string.encode('utf8')
tarinfo = tarfile.TarInfo(name='foo.csv')
tarinfo.size = len(file_data)
tarinfo.mtime = time.time()
tar.addfile(tarinfo, BytesIO(file_data))
tar.close()
```

Note that we've now created a csv file of the first 10 samples and converted it to a tarstream.

We can then use the `docker-py` package to directly connect to docker from our Python console, allowing us to extract the archived csv file directly in our shared storage using the `put_archive()` command. This will only work when the instructions in section B.3 have been followed, and the cluster is currently up and running.

```
client = docker.from_env()

spark_master = client.containers.get("spark-master")
```

[3]https://github.com/RoelBouman/ArtificialStreaming
[4]https://gist.github.com/zbyte64/6800eae10ce082bb78f0b7a2cca5cbc2

```
spark_worker_1 = client.containers.get("spark-worker-1")
spark_worker_2 = client.containers.get("spark-worker-2")

tarstream.seek(0)
spark_master.put_archive("/opt/spark-data", tarstream)
```

We can now indeed find the csv file written to the shared storage of our cluster.

B.4.2 Streaming data periodically to the cluster

The procedure for writing data can be modified to do so periodically. The example code below will for example write 10 measurements every 5 seconds to the shared storage of the cluster. We will use the `timeloop` package[5] in order to periodically perform jobs. The self-contained script which can read and write the data canis also be found on GitHub[6] as `readh5_writecsv.py`. It should be noted that we could construct a simpler artificial streaming pipeline by writing the data in partitions to disk from the script that also analyzes it, and subsequently setting the batch trigger to only handle 1 CSV file at a time, as is described in the Databricks documentation[7]. This approach is however less realistic, and not as extensible as our approach.

```
#%% Import require packages
import h5py
import numpy as np
import tarfile
import docker
import time
from timeloop import Timeloop
from datetime import timedelta
from io import BytesIO

#%% Loading data
filename = 'L701913_SAP000_B000_S0_P000_bf.h5'

h5 = h5py.File(filename, "r")

#%% Writing data to csv and tar
stokes = h5["/SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0"]

client = docker.from_env()
spark_master = client.containers.get("spark-master")

tl = Timeloop()

measurement_index = 0

@tl.job(interval=timedelta(seconds=5))
def read_and_write():
```

[5]https://github.com/sankalpjonn/timeloop
[6]https://github.com/RoelBouman/ArtificialStreaming
[7]https://docs.databricks.com/applications/machine-learning/mllib/mllib-pipelines-and-stuctured-streaming.html

```
    global measurement_index
    data_part = stokes[measurement_index:(measurement_index+10),:]
    np.savetxt("tempdata/foo.csv", data_part, delimiter=",")

    data_part_string = open("tempdata/foo.csv", "r").read()

    tarstream = BytesIO()
    tar = tarfile.TarFile(fileobj=tarstream, mode='w')
    file_data = data_part_string.encode('utf8')
    tarinfo = tarfile.TarInfo(name="measurement"+str(measurement_index
     ↪  )+"-"+str(measurement_index+10)+".csv")
    tarinfo.size = len(file_data)
    tarinfo.mtime = time.time()
    tar.addfile(tarinfo, BytesIO(file_data))
    tar.close()

    tarstream.seek(0)
    spark_master.put_archive("/opt/spark-data/beamformed", tarstream)

    measurement_index=measurement_index+10
    print("Written"+"measurement"+str(measurement_index)+"-"+str(measu
     ↪  rement_index+10))

tl.start(block=True)
```

With the streaming back-end operational, we can now work on analyzing data using Spark Structured Streaming from a CSV data source. Do not forgot to clear data you no longer need. This can easily be done using the following command:

```
rm /opt/spark-data/beamformed/*
```

from the base directory inside the docker container after executing:

```
docker exec -it spark-master /bin/bash
```

One can also remove these files directly from the mount on the local disk by running:

```
rm /mnt/spark-data/beamformed/*
```

In addition, we also provide a simple script, clearing all data and results in the shared memory. This script can be found on Github[8], and can be run by using the following command:

```
./clear_shared_memory_docker.sh
```

In order to execute the shell script, one must first modify the file permissions to allow execution:

```
chmod +x clear_shared_memory_docker.sh
```

The contents of the file are simple, and can be found below:

[8]https://github.com/RoelBouman/useful_scripts_thesis

```
#!/bin/bash

rm -f /mnt/spark-data/beamformed/*
rm -f /mnt/spark-data/weather/*
rm -f /mnt/spark-data/metadata/*

rm -f /mnt/spark-results/medians/*
rm -f /mnt/spark-results/median_scaled_data/*
rm -f /mnt/spark-results/power_filter/*
```

B.4.3 Basic streaming operations on a CSV data source

Now that we have an active process generating CSV files in the shared storage of our Spark cluster we can start analyzing it. In order to verify that everything works we can first run a simple application which will calculate and update the mean of the first variable in the dataset we are streaming. The code can be found below, or on GitHub[9] as `analyze_basic_csvstream.py`.

```
from __future__ import print_function

from pyspark.sql import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.types import StructField
from pyspark.sql.types import StructType
from pyspark.sql.functions import mean


if __name__ == "__main__":


  spark = SparkSession \
    .builder \
    .appName("BasicCSVStreamAnalyzer") \
    .getOrCreate()

  # Create DataFrame representing the stream of CSVs
  # We define the schema for this test simply to be doubles for the
  ↪  960 entries.

  userSchema = StructType([StructField("V"+str(i), DoubleType(),
  ↪  False) for i in range(0,960)])
  csvDF = spark \
    .readStream \
    .option("sep", ",") \
    .schema(userSchema) \
    .csv("/opt/spark-data/beamformed")

  meanV1 = csvDF.groupBy().mean("V1")
```

[9] https://github.com/RoelBouman/ArtificialStreaming

```
    # Start running the query that prints the running counts to the
    ↪   console
  query = meanV1 \
    .writeStream \
    .outputMode('complete') \
    .format('console') \
    .start()

  query.awaitTermination()
```

This app can be distributed to the cluster by running:

```
docker cp analyze_basic_csvstream.py spark-master:opt/spark-apps
```

from the folder where the script is located, followed by:

```
./spark/bin/spark-submit --master spark://spark-master:7077
 ↪  opt/spark-apps/analyze_basic_csvstream.py
```

from the main directory of the Spark master. When we run the application we must also activate the CSV stream by running the `readh5_writecsv.py` script as outlined in Appendix B.4.2.

This will output the average of the first variable in our data, keep it updated, and print it on every trigger. The output of the first 16 batch triggers looks like this:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.22142751162368E13|
+--------------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+------------------+
|           avg(V1)|
+------------------+
|3.221698904064E13|
+------------------+

-------------------------------------------
Batch: 2
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.199081688970971...|
+--------------------+

-------------------------------------------
```

```
Batch: 3
-------------------------------------------
+-----------------+
|          avg(V1)|
+-----------------+
|3.1763560136704E13|
+-----------------+

-------------------------------------------
Batch: 4
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.168772167943964...|
+--------------------+

-------------------------------------------
Batch: 5
-------------------------------------------
+-------------------+
|            avg(V1)|
+-------------------+
|3.15300287348736E13|
+-------------------+

-------------------------------------------
Batch: 6
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.134104271942749...|
+--------------------+

-------------------------------------------
Batch: 7
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.119580371572053...|
+--------------------+

-------------------------------------------
Batch: 8
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.111163205335827...|
+--------------------+

-------------------------------------------
Batch: 9
-------------------------------------------
+------------------+
|           avg(V1)|
+------------------+
|3.1016590770176E13|
+------------------+

-------------------------------------------
Batch: 10
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.089915400508757...|
+--------------------+

-------------------------------------------
Batch: 11
-------------------------------------------
+------------------+
|           avg(V1)|
+------------------+
|3.0797545603072E13|
+------------------+

-------------------------------------------
Batch: 12
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.078852043233882...|
+--------------------+

-------------------------------------------
Batch: 13
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.076570111308231...|
+--------------------+

-------------------------------------------
Batch: 14
-------------------------------------------
+-------------------+
|            avg(V1)|
+-------------------+
|3.07172042866688E13|
+-------------------+

-------------------------------------------
Batch: 15
-------------------------------------------
+--------------------+
|             avg(V1)|
+--------------------+
|3.069190166544384E13|
+--------------------+
```

B.4.4 Extending the streaming pipeline to multiple datastreams

One of the major advantages of using Spark for online data analysis is its SQL functionality. One of the major features of Spark Structured Streaming is its ability to join datastreams in an online fashion. This can for example be useful when streams from different sources need to be combined for a later purpose such as classification.

In order to later demonstrate this online functionality we must first simulate two separate incoming datastreams. We will do this by adding a second datastream to the single beamformed datastream covered in Appendix B.4.2. This second datastream will consist of weather data gathered by the KNMI on their Hoogeveen station, located close to the main LOFAR antenna hub in Dwingeloo. The KNMI offers hourly weather data per station through an API.

We will cover the main points of consideration during the development of this pipeline and will offer a listing of the code further below in this section. Firstly we can see that the periodic job ran through the `Timeloop` package is extended. In addition to only streaming a CSV to the cluster with numeric data, we will add two extra columns containing timestamp information needed for any SQL join operation. We add both a timestamp and an hourly timestamp. The first we use for tracking the time of each sample, which is useful for any practical purpose, while we use the second for joining with the weather data, which is only gathered each hour. We could theoretically also convert this timestamp from within the analysis pipeline, but the timestamp datatype in PySpark is less flexible than that used in native Python.

After constructing the dataframe of beamformed data we check if we can gather new weather data. As the data can only be gathered each hour and the job runs each 5 seconds we don't want overhead caused by streaming duplicate data. We could alternatively establish two separate jobs with different intervals, but this would be designed in a manner less close to any practical application. In addition, this flexible approach works when one wants to experiment with different batch sizes or time intervals in an intuitive and easy manner. When new weather data should be gathered it is acquired through a Python wrapper for the API, called `KNMY`[10]. The data is cleaned in order to be written and read as a CSV and a timestamp is given. We also parse the timestamp so it can be used as a watermark variable in the CSV, as well as using it in the filename. The CSV files are then written to different folders on the Spark cluster, representing different incoming streams. It should lastly be noted that the microseconds in timestamps are dropped when converting to CSV, as these are

[10] https://pypi.org/project/knmy/

not natively parsed by Spark. This results in loss of time resolution which may need to be resolved when samples need to be ordered.

The code below implements the procedure described above, and can in addition be found on GitHub[11] as `readh5_writecsv_beamformed_and_weather.py`.

```
#%% Import require packages
import h5py
import pandas as pd
import tarfile
import docker
import time
import re
from knmy import knmy
from timeloop import Timeloop
from datetime import timedelta
from dateutil.parser import parse
from io import BytesIO

#%% Helper functions.

def to_writeable_timestamp(timestamp):
    timestamp_string = str(timestamp)

    cleaned_timestamp_string = re.sub(r'[ :]', '-', re.sub(r'\+.*',
     ↪  '',timestamp_string))

    return(cleaned_timestamp_string)

def write_beamformed(data_part_df):
    data_part_string = data_part_df.to_csv(sep=",",
     ↪  date_format="%Y-%m-%d %H:%M:%S", index=False)

    tarstream = BytesIO()
    tar = tarfile.TarFile(fileobj=tarstream, mode='w')
    file_data = data_part_string.encode('utf8')
    tarinfo = tarfile.TarInfo(name="measurement"+str(measurement_index ⌋
     ↪  )+"-"+str(measurement_index+index_delta)+".csv")
    tarinfo.size = len(file_data)
    tarinfo.mtime = time.time()
    tar.addfile(tarinfo, BytesIO(file_data))
    tar.close()

    tarstream.seek(0)
    spark_master.put_archive("/opt/spark-data/beamformed", tarstream)

def fetch_and_write_weather(last_timestamp, last_hourly_measurement):

    #Correct for datetime handling in knmy function by adding hour
     ↪  offset
```

[11] https://github.com/RoelBouman/ArtificialStreaming

```
    _, _, _, knmi_df = knmy.get_hourly_data(stations=[279],
     ↪  start=last_hourly_measurement-timedelta(hours=1),
     ↪  end=last_timestamp-timedelta(hours=1), parse=True)
    knmi_df = knmi_df.drop(knmi_df.index[0]) #drop first row, which
     ↪  contains a duplicate header

    knmi_df["timestamp"] = [(parse(date) + timedelta(hours=int(hour)))
     ↪  for date, hour in zip(knmi_df["YYYYMMDD"], knmi_df["HH"])]
    knmi_df = knmi_df.drop(["STN", "YYYYMMDD", "HH"], axis=1)

    weather_string = knmi_df.to_csv(sep=",", date_format="%Y-%m-%d
     ↪  %H:%M:%S", index=False)

    filename = "weather"+to_writeable_timestamp(last_hourly_measuremen ⌋
     ↪  t)+"-to-"+to_writeable_timestamp(last_timestamp)+".csv"
    tarstream = BytesIO()
    tar = tarfile.TarFile(fileobj=tarstream, mode='w')
    file_data = weather_string.encode('utf8')
    tarinfo = tarfile.TarInfo(name=filename)
    tarinfo.size = len(file_data)
    tarinfo.mtime = time.time()
    tar.addfile(tarinfo, BytesIO(file_data))
    tar.close()

    tarstream.seek(0)
    spark_master.put_archive("/opt/spark-data/weather", tarstream)

#%% initialize variables and initialize hdf5 access and docker
 ↪  containers
filename = 'L701913_SAP000_B000_S0_P000_bf.h5'

h5 = h5py.File(filename, "r")
stokes = h5["/SUB_ARRAY_POINTING_000/BEAM_000/STOKES_0"]

client = docker.from_env()
spark_master = client.containers.get("spark-master")

tl = Timeloop()

time_start = parse(h5.attrs['OBSERVATION_START_UTC']) #Start of
 ↪  measurements as datetime object
measurement_index = 102984 #Which measurement should streaming start
 ↪  with?
# For when to test the hourly change:
# 18*60*(10**6)/time_delta.microseconds
# Out[172]: 102994.46881556361
# with 102984 first batch should have weather from hour 11, second
 ↪  batch from hour 12
index_delta = 100 #How many measurements should be sent each second
```

```
time_delta = timedelta(seconds=h5["/SUB_ARRAY_POINTING_000/BEAM_000/CO⌋
↪  ORDINATES/COORDINATE_0"].attrs["INCREMENT"]) #The time between two
↪  consecutive measurements


# calculate first timestamp and set it to 1 hour before measurements
↪  start
last_hourly_measurement = (time_start -
↪  timedelta(hours=1)).replace(minute=0, second=0, microsecond=0)
#%% define jobs
@tl.job(interval=timedelta(seconds=5))
def read_and_write():
    global measurement_index
    global last_hourly_measurement

    #measurements -and- timestamp
    data_part_df = pd.DataFrame(stokes[measurement_index:(measurement_⌋
    ↪  index+index_delta),:])
    data_part_df["timestamp"] = [time_start+dt for dt in [d*time_delta
    ↪  for d in range(measurement_index,
    ↪  measurement_index+index_delta)]]
    data_part_df["hourly_timestamp"] =
    ↪  [(time_start+dt).replace(minute=0,second=0, microsecond=0) for
    ↪  dt in [d*time_delta for d in range(measurement_index,
    ↪  measurement_index+index_delta)]]


    #Determine if new weather data should be written
    if any(t.replace(minute=0, second=0, microsecond=0) >
    ↪  last_hourly_measurement for t in data_part_df["timestamp"]):
        #write weather data
        last_timestamp =
        ↪  data_part_df["timestamp"].iloc[-1].replace(minute=0,
        ↪  second=0, microsecond=0)
        fetch_and_write_weather(last_timestamp,
        ↪  last_hourly_measurement)

        last_hourly_measurement = last_timestamp

    #write beamformed data
    write_beamformed(data_part_df)

    print("Written measurements "+str(measurement_index)+"-"+str(measu⌋
    ↪  rement_index+index_delta))
    measurement_index=measurement_index+index_delta

#%% start jobs
tl.start(block=True)
```

B.4.5 Basic operations on joined datastreams

With multiple datastreams being written to the cluster, we can join them using Structured Streaming, and perform basic operations on them. Much like outlined in Appendix B.4.3, we will first need to execute the script described in Appendix B.4.4. When the script writing the CSVs to the shared storage is running we can execute our job. Alternatively, the Spark job can be ran before the writing script in anticipation, in order to follow execution from beginning to end.

The Python script submit to Spark can be found on GitHub[12] as `analyze_dual_csvstream.py`. In addition, a listing of the code is found at the end of this section. We will concisely describe the script below.

When defining the stream, we now take into account the newly added timestamp columns on the beamformed data. In addition, we also define a CSV-stream on the newly streamed weather data. The data consists out of 22 doubles describing different aspects of the weather at Hoogeveen as measured by the KNMI, and also consists of an added timestamp.

After defining the streams we set the timestamp columns as watermarks. These watermarks are used by Spark in order to determine in which windows ordered samples should be handled, as well as define a time after which to disregard data coming in late.

The watermarked dataframes are then joined using a left outer join so that the weather data is joined with the beamformed data according to a matching timestamp. We need to define an interval after which we will disregard late incoming data, otherwise the join operation can not be performed in a streaming manner.

We then perform a sample operation on the data in order to ensure that the joining operation has succeeded. We will perform a simply aggregation on the maximum beamformed timestamp, and show the corresponding weather timestamp.

```
from __future__ import print_function

from pyspark.sql import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.types import StructField
from pyspark.sql.types import StructType
from pyspark.sql.types import TimestampType
from pyspark.sql.functions import col
from pyspark.sql.functions import max as max_
from pyspark.sql.functions import expr


if __name__ == "__main__":


  spark = SparkSession \
    .builder \
    .appName("DualCSVStreamAnalyzer") \
    .getOrCreate()

  # Create DataFrame representing the stream of CSVs
```

[12]https://github.com/RoelBouman/ArtificialStreaming

```
# We define the schema for this test simply to be doubles for the
↪  960 entries.
# The last entry is a timestamp

beamformedFieldTypes = [StructField("V"+str(i), DoubleType(), False)
↪  for i in range(0,960)]
beamformedFieldTypes.append(StructField("beamformedtimestamp",
↪  TimestampType(), False))
beamformedFieldTypes.append(StructField("hourly_beamformedtimestamp"⌋
↪  , TimestampType(),
↪  False))
beamformedSchema = StructType(beamformedFieldTypes)
beamformedDF = spark \
  .readStream \
  .option("sep", ",") \
  .option("header", "true") \
  .schema(beamformedSchema) \
  .csv("/opt/spark-data/beamformed")

weatherFieldTypes = [StructField("V"+str(i), DoubleType(), True) for
↪  i in range(0,22)]
weatherFieldTypes.append(StructField("weathertimestamp",
↪  TimestampType(), False))
weatherSchema = StructType(weatherFieldTypes)
weatherDF = spark \
  .readStream \
  .option("sep", ",") \
  .option("header", "true") \
  .schema(weatherSchema) \
  .csv("/opt/spark-data/weather")

# Watermark data
watermarkedBeamformedDF =
↪  beamformedDF.withWatermark("beamformedtimestamp", "1 hours")
watermarkedWeatherDF = weatherDF.withWatermark("weathertimestamp",
↪  "2 hours")

# Joining dataframes
joinedDF = watermarkedBeamformedDF.join(
  watermarkedWeatherDF,
  expr("""
    hourly_beamformedtimestamp = weathertimestamp AND
    beamformedtimestamp >= weathertimestamp AND
    beamformedtimestamp <= weathertimestamp + interval 1 hour
    """),
  "leftOuter")

# Sample operation on joined data
maxTimeStamp = joinedDF.groupby("beamformedtimestamp",
↪  "weathertimestamp").agg(max_("beamformedtimestamp"))
```

```
  # Start running the query that prints the running counts to the
   ↪  console
  query = maxTimeStamp \
    .writeStream \
    .outputMode('complete') \
    .format('console') \
    .start()

  query.awaitTermination()
```

To reproduce the experiment we first need to start our spark cluster and initialize the analysis pipeline by performing the following commands:

```
docker-compose up -d

docker exec -it spark-master /bin/bash

./spark/bin/spark-submit --master spark://spark-master:7077
 ↪  /opt/spark-apps/analyze_dual_csvstream.py
```

followed by initializing the plotting procedure from a separate console:

```
python readh5_writecsv_beamformed_and_weather.py
```

After execution, the remaining CSV data can be cleaned by executing:

```
rm /opt/spark-data/beamformed/*
rm /opt/spark-data/weather/*
```

From within the `spark-master` container, or by running the helper script from outside the container:

```
./clear_shared_memory_docker.sh
```

The results of this pipeline will in practice be used as an intermediate for a complete pipeline, or they will be written to disk. To illustrate that the procedure works, we have instead performed a groupby operation on the beamformed and weather timestamps, followed by an aggregation where we select the maximum beamformed timestamp. This aggregation is largely redundant, but allows for a simple console output of two identifying columns of the resulting joined dataframe. The sample output, for the first few microbatches, can be found below:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+------------------------+


-------------------------------------------
```

```
Batch: 1
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+------------------------+

-------------------------------------------
Batch: 2
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+------------------------+

-------------------------------------------
Batch: 3
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+------------------------+

-------------------------------------------
Batch: 4
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
+-------------------+-------------------+------------------------+

-------------------------------------------
Batch: 5
-------------------------------------------
+-------------------+-------------------+------------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+------------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 6
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
|2019-04-13 12:00:17|2019-04-13 12:00:00|     2019-04-13 12:00:17|
+-------------------+-------------------+-----------------------+

-------------------------------------------
Batch: 7
-------------------------------------------
+-------------------+-------------------+-----------------------+
|beamformedtimestamp|   weathertimestamp|max(beamformedtimestamp)|
+-------------------+-------------------+-----------------------+
|2019-04-13 12:00:05|2019-04-13 12:00:00|     2019-04-13 12:00:05|
|2019-04-13 12:00:11|2019-04-13 12:00:00|     2019-04-13 12:00:11|
|2019-04-13 12:00:14|2019-04-13 12:00:00|     2019-04-13 12:00:14|
|2019-04-13 12:00:13|2019-04-13 12:00:00|     2019-04-13 12:00:13|
|2019-04-13 12:00:00|2019-04-13 12:00:00|     2019-04-13 12:00:00|
|2019-04-13 12:00:04|2019-04-13 12:00:00|     2019-04-13 12:00:04|
|2019-04-13 12:00:16|2019-04-13 12:00:00|     2019-04-13 12:00:16|
|2019-04-13 12:00:08|2019-04-13 12:00:00|     2019-04-13 12:00:08|
|2019-04-13 12:00:19|2019-04-13 12:00:00|     2019-04-13 12:00:19|
|2019-04-13 12:00:02|2019-04-13 12:00:00|     2019-04-13 12:00:02|
|2019-04-13 12:00:07|2019-04-13 12:00:00|     2019-04-13 12:00:07|
|2019-04-13 12:00:10|2019-04-13 12:00:00|     2019-04-13 12:00:10|
|2019-04-13 12:00:06|2019-04-13 12:00:00|     2019-04-13 12:00:06|
|2019-04-13 12:00:12|2019-04-13 12:00:00|     2019-04-13 12:00:12|
|2019-04-13 12:00:03|2019-04-13 12:00:00|     2019-04-13 12:00:03|
|2019-04-13 12:00:09|2019-04-13 12:00:00|     2019-04-13 12:00:09|
|2019-04-13 12:00:01|2019-04-13 12:00:00|     2019-04-13 12:00:01|
|2019-04-13 11:59:59|2019-04-13 11:00:00|     2019-04-13 11:59:59|
|2019-04-13 12:00:15|2019-04-13 12:00:00|     2019-04-13 12:00:15|
|2019-04-13 12:00:18|2019-04-13 12:00:00|     2019-04-13 12:00:18|
+-------------------+-------------------+------------------------+
only showing top 20 rows
```

Appendix C

Streaming Applications for ASTRON

C.1 Current beamformed analysis at ASTRON

Currently at ASTRON, beamformed data from solar observations is being processed in an online fashion using a Python script. This script reads the observations from a `.raw` file connected to a layer of hdf5 information in the form of a `.h5` layer. This file structure is similar to that elaborated upon in appendices A and B. The major difference is that the `.raw` file grows as new beamformed observations are being written to disk at the CEP.

The script then reads from the `.raw` file as a buffer, and processes each read, with a maximum of 10000 samples per read. The numerical data is normally subsampled 1 in every 25 measurements over the time axis. Every one of these batches, with a maximum size of 400-by-the number of frequencies measured, is then scaled by dividing it by the median of this batch, a procedure called normalization by ASTRON. Each of these scaled batches is appended to a larger matrix-like data structure which is visualized for each iteration of this procedure. The code as used by ASTRON is provided under the Apache 2.0 license and can be found below as well as on Github[1]

```
#! /usr/bin/env python
import sys
from argparse import ArgumentParser
import os
import h5py
import matplotlib.pyplot as plt
import numpy as np
import casacore.tables as pt
import matplotlib.dates as mdates
from datetime import datetime as dt


parser = ArgumentParser("create dynspectrum plot from raw data");
parser.add_argument('-i','--rawdata', nargs='+',help='list of rawdata
 ↪  files',dest="rawfile",required=True)
parser.add_argument('-k','--skip_samples', help='number of samples to
 ↪  skip',dest='skip_samples',type=int,default=0)
parser.add_argument('-s','--vmin', help='vmin of
 ↪  plot',dest='vmin',type=float,default=0.5)
```

[1] https://github.com/maaijke/ROscripts

```
parser.add_argument('-m','--vmax', help='vmax of
 ↪  plot',dest='vmax',type=float,default=2)
parser.add_argument('-c','--cmap', help='matplotlib
 ↪  colormap',dest='cmap',default="viridis")
parser.add_argument('-n','--show_normalization', help='plot
 ↪  normalization',dest='show_normalization', action='store_true')

#To DO: fix complex_voltages plotting (make sure they start at the
 ↪  same time). Save as png. Do not keep full buffer in memory. Get
 ↪  info from h5files. Maybe keep x-axis fixed. # plotmedian #add axes
 ↪  etc #find parset# figure ut how to stop the plotting

bytes_per_sample=4

def get_metadata_from_h5(h5file):
     #metadata=h5file.attrs[u'NOF_SUB_ARRAY_POINTINGS']
     metadata=dict(h5file[h5file.visit(lambda x: x if 'STOKES' in x
      ↪  else None)].attrs)
     metadata['freqs'] = h5file[h5file.visit(lambda x: x if
      ↪  'COORDINATE_1' in x else None)].attrs[u'AXIS_VALUES_WORLD']
     metadata=dict(metadata,**dict(h5file[h5file.visit(lambda x: x if
      ↪  'BEAM' in x else None)].attrs))
     metadata["starttime"]= h5file.attrs[u'OBSERVATION_START_MJD']
     metadata["endtime"]= h5file.attrs[u'OBSERVATION_END_MJD']
     return metadata


def plot_real_time(fig,axarr,rawfile,nch,nSB,freqs,vmin,vmax,maxSample ⌋
 ↪  s=10000,skiptime=25,skipch=1,skipSamples=0,starttime=None,endtime= ⌋
 ↪  None,sampleSize=1./125.,cmap='Reds',show_norm=False):
     hasdata=False
     rawfile.seek(skipSamples*bytes_per_sample*nch*nSB)
     if not starttime is None:
          starttime += (skipSamples*sampleSize)/(24*3600.)
          starttime_dt =dt.strptime(pt.taql('select str(mjdtodate({}))
           ↪  as date'.format(starttime)).getcol('date')[0],
           ↪  '%Y/%m/%d/%H:%M:%S')
          fig.suptitle(starttime_dt.strftime("%m/%d/%Y"))
     while(True):
        mybuffer = rawfile.read(maxSamples*bytes_per_sample*nch*nSB)
        tmpdata=np.frombuffer(mybuffer,dtype=np.float32) #np.float =
         ↪  np.float64!!
        nSam=tmpdata.shape[0]/(nch*nSB)
        if not hasdata:
             data=tmpdata.reshape(nSam,(nch*nSB))[::skiptime,::skipch]
             hasdata=True
        else:
             data=np.concatenate((data,tmpdata.reshape(nSam,(nch*nSB)) ⌋
              ↪  [::skiptime,::skipch]),axis=0)
        mymedian=np.median(data,axis=0)
        #fig.clf()
        ax=axarr
        if show_norm:
            ax=ax[0]
        ax.cla()
        myextent=[0,data.shape[0],freqs[0]*1e-6,freqs[::skipch][-1]*1e
        ↪ -6]
        if not (starttime is None):
            myextent[0]=mdates.date2num(starttime_dt)
            myextent[1]=mdates.date2num(dt.strptime(pt.taql('select
              ↪ str(mjdtodate({})) as
              ↪ date'.format(starttime+(data.shape[0]*skiptime*sample
              ↪ Size)/(24.*3600.))).getcol('date')[0],
              ↪ '%Y/%m/%d/%H:%M:%S'))  # thanks to TJD
            ax.imshow((data/mymedian).T,origin='lower',interpolation=
              ↪ 'nearest',aspect='auto',vmin=vmin,vmax=vmax,extent=my
              ↪ extent,cmap=cmap)
        ax.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
        ax.set_ylabel("freq (MHz)")
        if show_norm:
            ax=axarr[1]
            ax.cla()
            ax.plot(freqs[::skipch]*1e-6,mymedian,'k')
            ax.set_xlabel("freq (MHz)")
        plt.pause(.3)
        #savefig("fig.png")


def plot_real_time_complex_voltages(rawfiles,nch,nSB,freqs,vmin,vmax,m
↪ axSamples=10000,skiptime=100,skipch=1):
    hasdata=False
    buffers=['','','','']
    prevminsize=0
    while True:
        minsize=-1
        for i in xrange(len(rawfiles)):
            buffers[i] = buffers[i][prevminsize:]+rawfiles[i].read(max
             ↪ Samples*bytes_per_sample*nch*nSB)
            #print minsize,len(buffers[i])
            if minsize<0 or len(buffers[i])<minsize:
                minsize=len(buffers[i])
        prevminsize=minsize
        tmpdata=np.frombuffer(buffers[0][:minsize],dtype=np.float32)
        nSam=tmpdata.shape[0]/(nch*nSB)
        print "reshaping",tmpdata.shape[0],"to",nSam,nch*nSB,minsize
        tmpdata=tmpdata.reshape(nSam,(nch*nSB))[::skiptime,::skipch]
        tmpdata= tmpdata**2 + \
              np.frombuffer(buffers[1][:minsize],dtype=np.float32).res
              ↪ hape(nSam,(nch*nSB))[::skiptime,::skipch]**2 +
              ↪ \
            np.frombuffer(buffers[2][:minsize],dtype=np.float32).res
             ↪   hape(nSam,(nch*nSB))[::skiptime,::skipch]**2 +
             ↪   \
            np.frombuffer(buffers[3][:minsize],dtype=np.float32).res
             ↪   hape(nSam,(nch*nSB))[::skiptime,::skipch]**2
        tmpdata=np.sqrt(tmpdata)
        if not hasdata:
             data=tmpdata
             hasdata=True
        else:
             data=np.concatenate((data,tmpdata),axis=0)
        mymedian=np.median(data,axis=0)
        clf()
        subplot(2,1,1)
        imshow((data/mymedian).T,origin='lower',interpolation='nearest
         ↪   ',aspect='auto',vmin=vmin,vmax=vmax,extent=(0,data.shape[0
         ↪   ],freqs[0]*1e-6,freqs[::skipch][-1]*1e-6))
        ylabel("freq (MHz)")
        subplot(2,1,2)
        plot(freqs[::skipch]*1e-6,mymedian,'k')
        xlabel("freq (MHz)")
        pause(.1)
        #savefig("fig.png")

def main(argv):
    args=parser.parse_args(argv)
    skipSamples=args.skip_samples
    rawfiles=[open(i,'rb') for i in args.rawfile]
    metadata = get_metadata_from_h5(h5py.File(args.rawfile[0].replace(
     ↪   '.raw','.h5')))
    #a=[rawfile.seek(-1000*60*16*4,os.SEEK_END) for rawfile in
     ↪   rawfiles]
    #buffers=[rawfile.read(1000*400*1*4) for rawfile in rawfiles]
    if not metadata[u'COMPLEX_VOLTAGE']:
            for i,rawfile in enumerate(rawfiles):
                 fig,axarr=plt.subplots(1+args.show_normalization,1)
                 plot_real_time(fig,axarr,rawfile,metadata['CHANNELS_P
                  ↪   ER_SUBBAND'],metadata[u'NOF_SUBBANDS'],metadata['
                  ↪   freqs'],args.vmin,args.vmax,skipSamples=skipSampl
                  ↪   es,starttime=metadata['starttime'],endtime=metada
                  ↪   ta['endtime'],sampleSize=metadata[u'SAMPLING_TIME
                  ↪   '],cmap=args.cmap,show_norm=args.show_normalizati
                  ↪   on)
    else:
            for i in range(len(rawfiles))[::4]:
                rawfile4=rawfiles[i*4:i*4+4]
                fig,axarr=plt.subplots(2,1)
                plot_real_time_complex_voltages(fig,axarr,rawfile4,met
                 ↪   adata['CHANNELS_PER_SUBBAND'],metadata[u'NOF_SUBBA
                 ↪   NDS'],metadata['freqs'],args.vmin,args.vmax)
```

```
if __name__ == '__main__':
    main(sys.argv[1:])
```

This script is normally run from the command line, the main argument being the raw file which should be analyzed and optional arguments controlling the visualization aspect of the analysis as well as the skipping of initial samples.

C.2 Adapting the beamformed analysis to stream to a Spark cluster

We will adapt the procedure developed for the analysis of beamformed data to stream data from a growing .raw file to our Spark cluster. This introduces an extra read and write step, which is not optimal in terms of processing time. Ideally, an alternative DataStreamReader input based on the .raw/.h5 file structure used by ASTRON should be developed and used. More details are discussed in section 5.

The developed pipeline is a simple adaptation of the script provided by ASTRON. Instead of processing the files in the script, the data which is to be processed is instead written to `.csv` files on the spark cluster. In addition to the raw data, metadata pertaining to the measurement is also written to the cluster to allow for optional processing based on this metadata. As ordering is not guaranteed in Spark, we will also add additional columns including the timestamp, and the seconds that have passed for each measurement since the start of the experiment.

Lastly, we have added several additional arguments to the command line interface, these are settings present in the original ASTRON script, but are now exposed to the user. In addition to these settings we have also introduced a wait time, and a subsampling ratio which can be used to balance the load on the Spark cluster during development.

The code can be found below, or on GitHub[2] as `read_binary_buffer_write_csvstream.py`.

```
#! /usr/bin/env python
import sys
from io import BytesIO
from argparse import ArgumentParser
import h5py
import numpy as np
import tarfile
import docker
import time
import pandas as pd
from astropy.time import Time as astroTime
from datetime import timedelta
import pickle

parser = ArgumentParser("create csvstream from raw data");
parser.add_argument('-i','--rawdata',help='rawdata
↪  file',dest="rawfile",required=True)
```

[2]https://github.com/RoelBouman/ArtificialStreaming

```
parser.add_argument('-d','--docker_container',help='docker
 ↪  container',dest="docker_container",type=str,required=True)
parser.add_argument('-fd','--docker_folder_data',help='docker folder
 ↪  for beamformed
 ↪  data',dest="docker_folder_data",type=str,required=True)
parser.add_argument('-fm','--docker_folder_metadata',help='docker
 ↪  folder for
 ↪  metadata',dest="docker_folder_metadata",type=str,required=True)
parser.add_argument('-k','--skip_samples', help='number of samples to
 ↪  skip at the beginning',dest='skip_samples',type=int,default=0)
parser.add_argument('-t','--skip_time', help='time sampling ratio
 ↪  (e.g. 25 for taking 1 in 25 samples, samples over all read
 ↪  samples)',dest='skip_time',type=int,default=25)
parser.add_argument('-m','--max_samples', help='number of samples to
 ↪  read per streaming operation, must be higher than
 ↪  skip_time',dest='max_samples',type=int,default=10000)
parser.add_argument('-w','--wait_time', help='time in seconds to wait
 ↪  between read/stream
 ↪  operations',dest='wait_time',type=int,default=0)

bytes_per_sample=4

def write_file_to_container(docker_container,docker_folder,data_string⌋
 ↪  ,file_name):

     tarstream=BytesIO()
     tar=tarfile.TarFile(fileobj=tarstream, mode='w')
     tarinfo=tarfile.TarInfo(name=file_name)
     tarinfo.size=len(data_string)
     tarinfo.mtime=time.time()
     tar.addfile(tarinfo, BytesIO(data_string))
     tar.close()

     tarstream.seek(0)
     docker_container.put_archive(docker_folder, tarstream)

def write_metadata(docker_container,docker_folder,metadata,file_name):
     pickled_metadata = pickle.dumps(metadata) #pickles metadata as
      ↪  string

     write_file_to_container(docker_container,docker_folder,pickled_me⌋
      ↪  tadata,file_name)

def write_beamformed(docker_container,docker_folder,data_part_df,start⌋
 ↪  time,endtime):
     data_string = data_part_df.to_csv(sep=",", date_format="%Y-%m-%d
      ↪  %H:%M:%S", index=False)
     data_string=data_string.encode('utf8')
     file_name="measurement"+str(np.round(starttime,
      ↪  4))+"-"+str(np.round(endtime,4))+".csv"
```

```
    write_file_to_container(docker_container,docker_folder,data_strin ⌋
     ↪  g,file_name)

def get_metadata_from_h5(h5file):
    metadata=dict(h5file[h5file.visit(lambda x: x if 'STOKES' in x
     ↪  else None)].attrs)
    metadata['freqs'] = h5file[h5file.visit(lambda x: x if
     ↪  'COORDINATE_1' in x else None)].attrs[u'AXIS_VALUES_WORLD']
    metadata=dict(metadata,**dict(h5file[h5file.visit(lambda x: x if
     ↪  'BEAM' in x else None)].attrs))
    metadata["starttime"]= h5file.attrs[u'OBSERVATION_START_MJD']
    metadata["endtime"]= h5file.attrs[u'OBSERVATION_END_MJD']
    return metadata

def stream_real_time(docker_container,docker_folder_data,rawfile,nch,n ⌋
 ↪  SB,freqs,starttime,endtime,maxSamples=10000,skiptime=25,skipch=1,s ⌋
 ↪  kipSamples=0,sampleSize=1./125.,waitTime=0):

    rawfile.seek(skipSamples*bytes_per_sample*nch*nSB)
    tSam = int(skipSamples)
    while(True):
        mybuffer=rawfile.read(maxSamples*bytes_per_sample*nch*nSB)
        tmpdata=np.frombuffer(mybuffer,dtype=np.float32) #np.float =
         ↪  np.float64!!
        nSam=int(tmpdata.shape[0]/(nch*nSB))
        tSam+=nSam

        data_part_df=pd.DataFrame(tmpdata.reshape((int(nSam),(nch*nSB) ⌋
         ↪  ))[::skiptime,::skipch])

        data_part_df["seconds_from_start"]=[d*sampleSize for d in
         ↪  range(int(tSam-nSam), int(tSam))][::skiptime]
        data_part_df["timestamp"]=[astroTime(starttime,format="mjd").d ⌋
         ↪  atetime+timedelta(seconds=dt) for dt in
         ↪  data_part_df["seconds_from_start"]]
        data_part_df["hourly_timestamp"]=[time.replace(minute=0,second ⌋
         ↪  =0, microsecond=0) for time in
         ↪  data_part_df["timestamp"]]

        write_beamformed(docker_container,docker_folder_data,data_part ⌋
         ↪  _df,(tSam-nSam)*sampleSize,(tSam*sampleSize))

        print(data_part_df.shape)
        print("streamed data from " +
         ↪  str(data_part_df["timestamp"].iloc[0]) + " to " +
         ↪  str(data_part_df["timestamp"].iloc[-1]) )

        time.sleep(waitTime)
```

```
def main(argv):

    args=parser.parse_args(argv)
    client=docker.from_env()
    docker_container=client.containers.get(args.docker_container)
    rawfile=open(args.rawfile,'rb')
    metadata = get_metadata_from_h5(h5py.File(args.rawfile.replace('.r
    ↪  aw','.h5')))

    write_metadata(docker_container,args.docker_folder_metadata,metada
    ↪  ta,args.rawfile.replace('.raw','_metadata.pickle'))

    stream_real_time(docker_container,args.docker_folder_data,rawfile,
    ↪  metadata['CHANNELS_PER_SUBBAND'],metadata[u'NOF_SUBBANDS'],met
    ↪  adata['freqs'],starttime=metadata['starttime'],endtime=metadat
    ↪  a['endtime'],skipSamples=args.skip_samples,sampleSize=metadata
    ↪  [u'SAMPLING_TIME'],maxSamples=args.max_samples,waitTime=args.w
    ↪  ait_time,skiptime=args.skip_time)


if __name__ == '__main__':
    main(sys.argv[1:])
```

One can then run this script to stream the data to a cluster as follows:

```
python read_binary_buffer_write_csvstream.py -i
↪  L701913_SAP000_B000_S0_P000_bf.raw -d spark-master -fd
↪  /opt/spark-data/beamformed -fm /opt/spark-data/metadata -m 500 -w
↪  5 -t 10
```

C.3 Processing the beamformed data stream using Spark Structured Streaming

In order to process the beamformed data stream, and perform the operations also performed by ASTRON some additional additions are needed to the cluster. Specifically we depend on Python 3.X, rather than Python 2.7.X, as several packages are no longer supported on the latter. We will also install the Cython, Numpy and Pandas libraries. In order to build these libraries on the Alpine base image we need to install several pieces of software in order to build these libraries on the system itself. This makes the system less lightweight, but does allow for the use of the Pandas library for the median scaling operations performed in subsections C.3.2 and C.3.3.

C.3.1 Online median calculation

In this section we will show how Spark Structured Streaming can be leveraged in order to replace the batch-wise median calculation with an online calculation using the Greenwald-Khanna algorithm [44]. Using Structured Streaming, we can express the median calculation as a windowed groupBy operation, followed by an aggregation calculation the approximate 50% percentile. The code can be found below, or on GitHub[3] as `streamingMedianAstronBeamformed.py`. In this example, we will only consider the first 3 variables to be able to still output the result to the console legibly.

```
from __future__ import print_function

from pyspark.sql import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.types import StructField
from pyspark.sql.types import StructType
from pyspark.sql.types import TimestampType
from pyspark.sql.functions import col
from pyspark.sql import DataFrame
from pyspark.sql.functions import window
from pyspark.sql.functions import expr


if __name__ == "__main__":


  spark = SparkSession \
    .builder \
    .appName("beamformedFiltering") \
    .getOrCreate()

  # Create DataFrame representing the stream of CSVs
  # We will define the schema based on the metadata
  # The last 3 entries consist of a the time in second from the start
  ↪  of the observation, the timestamp, and the timestamp with
  ↪  seconds and smaller time units dropped.

  beamformedFieldTypes = [StructField("V"+str(i), DoubleType(), False)
  ↪  for i in range(0,960)]
  beamformedFieldTypes.append(StructField("secondAfterMeasurement",
  ↪  DoubleType(), False))
  beamformedFieldTypes.append(StructField("beamformedTimestamp",
  ↪  TimestampType(), False))
  beamformedFieldTypes.append(StructField("hourlyBeamformedTimestamp",
  ↪  TimestampType(), False))
  beamformedSchema = StructType(beamformedFieldTypes)
  beamformedDF = spark \
    .readStream \
    .option("sep", ",") \
    .option("header", "true") \
    .schema(beamformedSchema) \
    .csv("/opt/spark-data/beamformed")

  exprs = [expr('percentile_approx('+x+', 0.5)').alias('med_'+x) for x
  ↪  in beamformedDF.columns[:3]]
```

[3]https://github.com/RoelBouman/ArtificialStreaming

```
medianDF = beamformedDF.withWatermark("beamformedTimestamp", "5
↪  seconds") \
    .groupBy(
      window("beamformedTimestamp", "5 seconds", "5 seconds") \
).agg(*exprs)

# Start running the query that prints the running counts to the
↪  console
query = medianDF \
    .writeStream \
    .outputMode('Append') \
    .format('console') \
    .start()

query.awaitTermination()
```

The code can be run easily by executing the following commands:

```
docker-compose up -d

docker exec -it spark-master /bin/bash

./spark/bin/spark-submit --master spark://spark-master:7077
↪  /opt/spark-apps/streamingMedianAstronBeamformed.py
```

Then the stream must also be initialized:

```
python read_binary_buffer_write_csvstream.py -i
↪  L701913_SAP000_B000_S0_P000_bf.raw -d spark-master -fd
↪  /opt/spark-data/beamformed -fm /opt/spark-data/metadata -m 500 -w
↪  5 -t 10
```

The partial output will look something like this, depending on the exact timing of micro-batch triggers:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------+------+------+------+
|window|med_V0|med_V1|med_V2|
+------+------+------+------+
+------+------+------+------+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+------+------+------+
|window|med_V0|med_V1|med_V2|
+------+------+------+------+
+------+------+------+------+

-------------------------------------------
Batch: 2
```

```
-------------------------------------------
+--------------------+------------+------------+------------+
|              window|      med_V0|      med_V1|      med_V2|
+--------------------+------------+------------+------------+
|[2019-04-13 11:42...|5.3764468E13|4.5321267E13|4.4340043E13|
|[2019-04-13 11:42...|6.0022986E13|6.4724973E13|6.6476283E13|
+--------------------+------------+------------+------------+

-------------------------------------------
Batch: 3
-------------------------------------------
+--------------------+------------+-----------+------------+
|              window|      med_V0|     med_V1|      med_V2|
+--------------------+------------+-----------+------------+
|[2019-04-13 11:42...|4.2138117E13|5.082905E13|5.0247183E13|
|[2019-04-13 11:42...|6.7746583E13|8.978774E13|  8.62959E13|
+--------------------+------------+-----------+------------+

-------------------------------------------
Batch: 4
-------------------------------------------
+--------------------+------------+------------+-----------+
|              window|      med_V0|      med_V1|     med_V2|
+--------------------+------------+------------+-----------+
|[2019-04-13 11:42...|3.0113253E13|3.0817932E13|3.182421E13|
+--------------------+------------+------------+-----------+

-------------------------------------------
Batch: 5
-------------------------------------------
+--------------------+------------+------------+------------+
|              window|      med_V0|      med_V1|      med_V2|
+--------------------+------------+------------+------------+
|[2019-04-13 11:42...|3.1559959E13|3.3749534E13|3.3770055E13|
+--------------------+------------+------------+------------+

-------------------------------------------
Batch: 6
-------------------------------------------
+------+------+------+------+
|window|med_V0|med_V1|med_V2|
+------+------+------+------+
+------+------+------+------+

-------------------------------------------
Batch: 7
-------------------------------------------
+--------------------+------------+------------+------------+
|              window|      med_V0|      med_V1|      med_V2|
+--------------------+------------+------------+------------+
|[2019-04-13 11:42...|2.8529327E13|2.9862184E13|3.0800754E13|
+--------------------+------------+------------+------------+

-------------------------------------------
Batch: 8
-------------------------------------------
+--------------------+------------+------------+------------+
|              window|      med_V0|      med_V1|      med_V2|
+--------------------+------------+------------+------------+
|[2019-04-13 11:42...|2.7716892E13|2.9159126E13|2.9177768E13|
+--------------------+------------+------------+------------+

-------------------------------------------
Batch: 9
-------------------------------------------
+------+------+------+------+
|window|med_V0|med_V1|med_V2|
+------+------+------+------+
+------+------+------+------+

-------------------------------------------
Batch: 10
-------------------------------------------
+--------------------+-----------+-----------+------------+
|              window|     med_V0|     med_V1|      med_V2|
+--------------------+-----------+-----------+------------+
|[2019-04-13 11:42...|3.802996E13|4.145049E13|3.8194834E13|
+--------------------+-----------+-----------+------------+
```

When running in the append output mode, only those for which all data is gathered are used in calculation. The windows in this example consist out of 5 seconds of measurements, and can be 5 seconds late before they are no longer considered in the calculation. The window is not based on machine time, but rather on the timestamp supplied with the data itself. Although the output is only written to the console in this example, it can also be written to a file output. In append mode, each line will only be written after the window timer has expired. The medians of all variables can also be calculated by modifying the expression given to the aggregation command.

C.3.2 Online median normalization

In the ASTRON application the beamformed data is normalized by dividing it through a windowed median. Unfortunately, this can not (yet) be expressed through the Structured Streaming API, as further operations, such as a join followed by a division on aggregated data, are not supported. Two workarounds exist to still formulate this procedure in Spark. The first option is to write the aggregated data to a stream, such as a Kafka stream, and import that back as an additional datasource so Spark no longer recognizes it as an aggregated stream. This does however incur substantial overhead due to the writing and reading of Kafka streams. The second option, which was implemented in this research, is instead of using one of the standard writers, using the forEachBatch writer, which allows arbitrary Python code to be used on each micro-batch. We can then calculate the median, and subsequently divide the beamformed data by it, for each micro-batch. This is substantially faster, and incurs less overhead than the previously mentioned workaround, but offers less control over the window over which the median is calculated. Effectively, we can no longer calculate the median as we did in subsection C.3.1 using a timestamp based window approach, but rather resort on batch-wise logic using the well known Python library Pandas. After the scaling operation, each batch can be written to disk, to allow for later analysis, or for a running plot of the data processed thus far. The code for the median scaling approach can be found below, or on GitHub[4] as medianScaleAstronBeamformed.py.

```
from __future__ import print_function

from pyspark.sql import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.types import StructField
from pyspark.sql.types import StructType
from pyspark.sql.types import TimestampType
from pyspark.sql.functions import col
from pyspark.sql import DataFrame
from pyspark.sql.functions import window
from pyspark.sql.functions import expr
from pyspark.sql.functions import hour
from pyspark.sql.functions import date_trunc


if __name__ == "__main__":


  spark = SparkSession \
    .builder \
    .appName("beamformedFiltering") \
    .getOrCreate()

  # Create DataFrame representing the stream of CSVs
  # We will define the schema based on the metadata
  # The last 3 entries consist of a the time in second from the start
  ↪  of the observation, the timestamp, and the timestamp with
  ↪  seconds and smaller time units dropped.

  variableNames = ["V"+str(i) for i in range(0,960)]

  beamformedFieldTypes = [StructField(v, DoubleType(), False) for v in
  ↪  variableNames]
  beamformedFieldTypes.append(StructField("secondAfterMeasurement",
  ↪  DoubleType(), False))
  beamformedFieldTypes.append(StructField("beamformedTimestamp",
  ↪  TimestampType(), False))
  #beamformedFieldTypes.append(StructField("hourlyBeamformedTimestamp"↩
  ↪  , TimestampType(),
  ↪  False))
  beamformedSchema = StructType(beamformedFieldTypes)
  beamformedDF = spark \
```

[4]https://github.com/RoelBouman/ArtificialStreaming

```
    .readStream \
    .option("sep", ",") \
    .option("header", "true") \
    .schema(beamformedSchema) \
    .csv("/opt/spark-data/beamformed") \
    .withWatermark("beamformedTimestamp", "5 seconds")

  def foreach_write(df, epoch_id):
    dataDF = df.select(variableNames).toPandas()
    bfTimestamp = df.select("beamformedTimestamp").toPandas()
    bfSecondsAfterMeasurement =
    ↪  df.select("secondAfterMeasurement").toPandas()

    writeColumns = variableNames + ["beamformedTimestamp"]

    median = dataDF.median()

    scaledDF = dataDF.divide(median)

    scaledDF["secondAfterMeasurement"] = bfSecondsAfterMeasurement
    scaledDF["beamformedTimestamp"] = bfTimestamp
    scaledDF = scaledDF.sort_values("secondAfterMeasurement")

    scaledDF.to_csv("/opt/spark-results/median_scaled_data/scaled_data
    ↪  " + str(epoch_id) + ".csv", header=True, index=False,
    ↪  columns=writeColumns)
    median.to_frame().T.to_csv("/opt/spark-results/medians/median" +
    ↪  str(epoch_id) + ".csv", header=True, index=False)

  query = beamformedDF.writeStream.foreachBatch(foreach_write).start()

  query.awaitTermination()
```

Now that the computationally relatively expensive scaling procedure can be performed on a Spark cluster, we can plot the results locally be reading the result files written by the cluster. To this end, we have developed a plotting procedure based on the ASTRON procedure, which can plot these results in an online manner. The code for this plotting procedure can be found below, or on GitHub[5] as plot_median_scaled_data.py.

To setup an end-to-end simulation of how the data is scaled, followed by plotting, the following commands must be executed:

Firstly set up the spark cluster:

```
docker-compose up -d

docker exec -it spark-master /bin/bash

./spark/bin/spark-submit --master spark://spark-master:7077
↪  /opt/spark-apps/medianScaleAstronBeamformed.py
```

followed by initializing the plotting procedure from a separate console:

[5]https://github.com/RoelBouman/ArtificialStreaming

```
python plot_median_scaled_data -r L701913_SAP000_B000_S0_P000_bf.raw
↪  -p /mnt/spark-results/median_scaled_data -m
↪  /mnt/spark-results/medians -n
```

Finally the stream can be initalized by issuing this command from a third console:

```
python read_binary_buffer_write_csvstream.py -i
↪  L701913_SAP000_B000_S0_P000_bf.raw -d spark-master -fd
↪  /opt/spark-data/beamformed -fm /opt/spark-data/metadata -m 500 -w
↪  5 -t 10
```

This will show an updating plot in an external window. A plot containing 4 intermediate results can be found in Figure C.1.

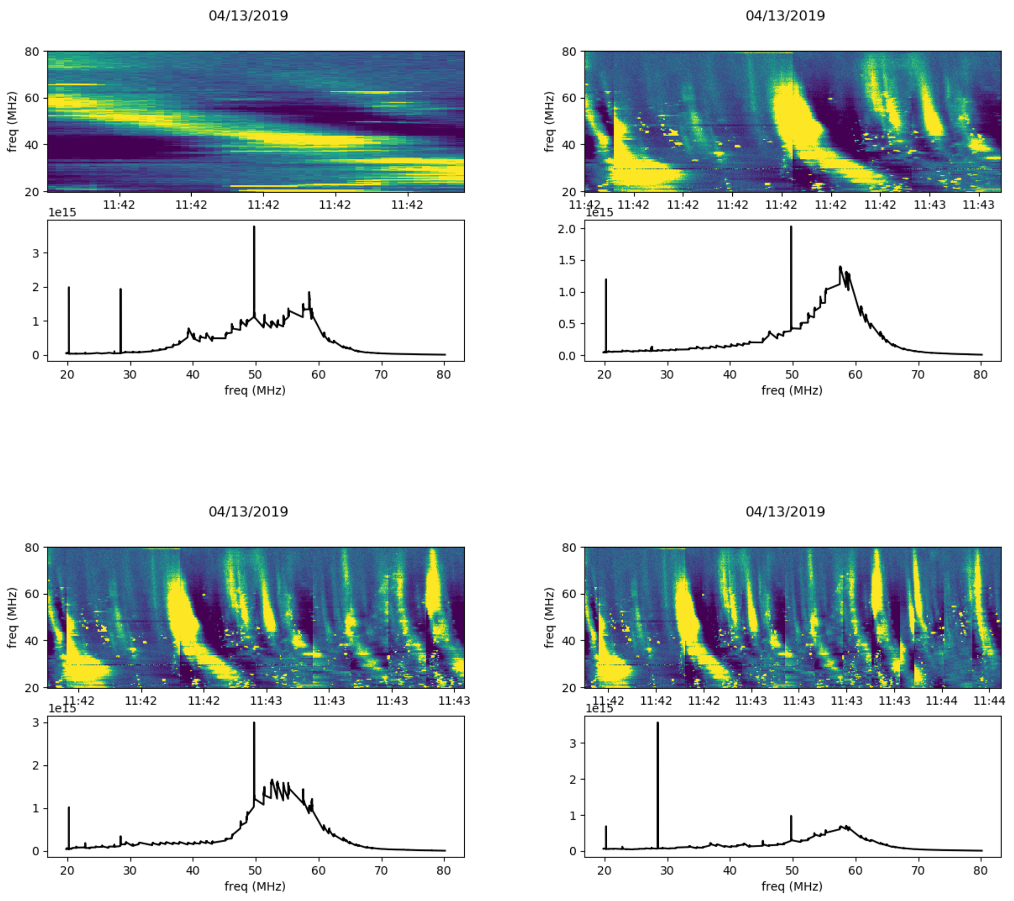

FIGURE C.1: Intermediate results from the online normalization procedure. 4 intermediate results are shown. The upper part of each subfigure shows the normalized beam-formed, or dynamic, spectrum, and the lower part shows the most recently calculated median, i.e. the one used to scale the most recent micro-batch. In the dynamic spectrum dark blue indicates a low intensity, whilst yellow indicates a high intensity.

C.3.3 Applying a power filter

As LOFAR has only recently been used to analyze space weather, the beamformed data is not yet labelled with the events that can be observed in it. As a preliminary scanning procedure, a power filter is often used to detect regions of interest in the spectrum. Afterwards, these regions of interest van be manually labelled, thus saving time in the labelling procedure. A power filter is a simple tool. One takes the sum of squares for each measured time point, so the sum of all the squared variables, and compares that to a threshold. Every time point with a sum of squares higher than the threshold is said to be a region of interest. In these experiments we have extended the procedure of the previous section to also include an online, so outside of the microbatch, calculation of the sum of squares. The code for the scaling approach with added power filter can be found below, or on GitHub[6] as `medianScalePowerFilterAstronBeamformed.py`.

```
from __future__ import print_function

from pyspark.sql import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.types import StructField
from pyspark.sql.types import StructType
from pyspark.sql.types import TimestampType
from pyspark.sql.functions import col
from pyspark.sql import DataFrame
#from pyspark.sql.functions import window
#from pyspark.sql.functions import expr
#from pyspark.sql.functions import hour
#from pyspark.sql.functions import date_trunc
from pyspark.sql.functions import udf, array


if __name__ == "__main__":


  spark = SparkSession \
    .builder \
    .appName("beamformedFiltering") \
    .getOrCreate()

  # Create DataFrame representing the stream of CSVs
  # We will define the schema based on the metadata
  # The last 3 entries consist of a the time in second from the start
  ↪  of the observation, the timestamp, and the timestamp with
  ↪  seconds and smaller time units dropped.

  variableNames = ["V"+str(i) for i in range(0,960)]

  beamformedFieldTypes = [StructField(v, DoubleType(), False) for v in
  ↪  variableNames]
```

[6]https://github.com/RoelBouman/ArtificialStreaming

```
beamformedFieldTypes.append(StructField("secondAfterMeasurement",
 ↪  DoubleType(), False))
beamformedFieldTypes.append(StructField("beamformedTimestamp",
 ↪  TimestampType(), False))
#beamformedFieldTypes.append(StructField("hourlyBeamformedTimestamp"⌋
 ↪  , TimestampType(),
 ↪  False))
beamformedSchema = StructType(beamformedFieldTypes)
beamformedDF = spark \
  .readStream \
  .option("sep", ",") \
  .option("header", "true") \
  .schema(beamformedSchema) \
  .csv("/opt/spark-data/beamformed") \
  .withWatermark("beamformedTimestamp", "5 seconds")

sumOfSquaresUdf = udf(lambda arr: sum(pow(a,2) for a in arr),
 ↪  DoubleType())

beamformedDF = beamformedDF.withColumn('sumOfSquares',
 ↪  sumOfSquaresUdf(array(variableNames)))


def foreach_write(df, epoch_id):
  dataDF = df.select(variableNames).toPandas()
  bfTimestamp = df.select("beamformedTimestamp").toPandas()
  bfSecondsAfterMeasurement =
   ↪  df.select("secondAfterMeasurement").toPandas()
  sumOfSquares = df.select("sumOfSquares").toPandas()

  writeColumns = variableNames + ['sumOfSquares',
   ↪  "beamformedTimestamp"]

  median = dataDF.median() #transpose to save each median in a
   ↪  separate column

  scaledDF = dataDF.divide(median)

  scaledDF["secondAfterMeasurement"] = bfSecondsAfterMeasurement
  scaledDF["beamformedTimestamp"] = bfTimestamp
  scaledDF["sumOfSquares"] = sumOfSquares
  scaledDF = scaledDF.sort_values("secondAfterMeasurement")

  scaledDF.to_csv("/opt/spark-results/median_scaled_data/scaled_data⌋
   ↪  " + str(epoch_id) + ".csv", header=True, index=False,
   ↪  columns=writeColumns)
  median.to_frame().T.to_csv("/opt/spark-results/medians/median" +
   ↪  str(epoch_id) + ".csv", header=True, index=False)

query = beamformedDF.writeStream.foreachBatch(foreach_write).start()
```

```
query.awaitTermination()
```

We have in addition extended the plotting procedure to also plot the power filter below the spectrum. We have not performed the thresholding procedure, and subsequent labelling, online, but rather leave it at the visualization phase. This allows the threshold to be chosen after the measurements have concluded. The code for this plotting procedure can be found below, or on GitHub[7] as `plot_median_scaled_data_with_power_filter.py`.

```
#%% Import require packages
import h5py
from argparse import ArgumentParser
import matplotlib.pyplot as plt
import os
import pandas as pd
import matplotlib.dates as mdates
import time
import sys
from dateutil.parser import parse
import re

parser = ArgumentParser("create dynspectrum plot from processed data");
parser.add_argument('-r','--rawdata',help='rawdata
↪  file',dest="rawfile",type=str,required=True)
parser.add_argument('-p','--processedata',help='processed data folder
↪  without trailing
↪  frontslash',type=str,dest="processedfolder",required=True)
parser.add_argument('-m','--processedmedian',help='processed median
↪  folder without trailing
↪  frontslash',type=str,dest="medianfolder",required=True)
parser.add_argument('-i','--vmin', help='vmin of
↪  plot',dest='vmin',type=float,default=0.5)
parser.add_argument('-a','--vmax', help='vmax of
↪  plot',dest='vmax',type=float,default=2)
parser.add_argument('-c','--cmap', help='matplotlib
↪  colormap',dest='cmap',default="viridis")
parser.add_argument('-n','--show_normalization', help='plot
↪  normalization',dest='show_normalization', action='store_true')
parser.add_argument('-w','--wait_time', help='wait
↪  time',dest='wait_time', default=5)
parser.add_argument('-f','--show_power_filter', help='plot power
↪  filter',dest='show_power_filter', action='store_true')
parser.add_argument('-t','--threshold', help='power filter
↪  threshold',dest='threshold', default=1e+32)


#https://stackoverflow.com/questions/5967500/how-to-correctly-sort-a-s
↪  tring-with-a-number-inside
def atof(text):
    try:
        retval = float(text)
```

[7] https://github.com/RoelBouman/ArtificialStreaming

```
    except ValueError:
        retval = text
    return retval

#https://stackoverflow.com/questions/5967500/how-to-correctly-sort-a-s
↪   tring-with-a-number-inside
def natural_keys(text):
    '''
    alist.sort(key=natural_keys) sorts in human order
    http://nedbatchelder.com/blog/200712/human_sorting.html
    (See Toothy's implementation in the comments)
    float regex comes from https://stackoverflow.com/a/12643073/190597
    '''
    return [ atof(c) for c in
    ↪   re.split(r'[+-]?([0-9]+(?:[.][0-9]*)?|[.][0-9]+)', text) ]

def get_metadata_from_h5(h5file):
     #metadata=h5file.attrs[u'NOF_SUB_ARRAY_POINTINGS']
     metadata=dict(h5file[h5file.visit(lambda x: x if 'STOKES' in x
     ↪   else None)].attrs)
     metadata['freqs'] = h5file[h5file.visit(lambda x: x if
     ↪   'COORDINATE_1' in x else None)].attrs[u'AXIS_VALUES_WORLD']
     metadata=dict(metadata,**dict(h5file[h5file.visit(lambda x: x if
     ↪   'BEAM' in x else None)].attrs))
     return metadata

def plot_real_time(fig,axarr,processed_data,freqs,vmin,vmax,median_dat
↪   a,maxSamples=10000,skiptime=25,skipch=1,sampleSize=1./125.,cmap='R
↪   eds',show_norm=False, show_power_filter=False,
↪   threshold=2e+17):
    ax=axarr

    starttime_dt = parse(processed_data.iloc[0,-1])
    endtime_dt = parse(processed_data.iloc[-1,-1])

    fig.suptitle(starttime_dt.strftime("%m/%d/%Y"))

    if show_norm | show_power_filter:
        ax=ax[0]
    ax.cla()
    myextent=[0,processed_data.shape[0],freqs[0]*1e-6,freqs[::skipch][
    ↪  -1]*1e-6]
    myextent[0]=mdates.date2num(starttime_dt)
    myextent[1]=mdates.date2num(endtime_dt)

    if show_power_filter:
        skip_cols = -2
    else:
        skip_cols = -1
    ax.imshow((processed_data.iloc[:,:skip_cols]).T,origin='lower',int⌋
    ↪  erpolation='nearest',aspect='auto',vmin=vmin,vmax=vmax,extent=⌋
    ↪  myextent,cmap=cmap)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
    ax.set_ylabel("freq (MHz)")
    if show_power_filter:
        ax=axarr[1]
        ax.cla()
        float_dates = [mdates.date2num(parse(date)) for date in
        ↪  processed_data.iloc[:,-1]]
        ax.plot(float_dates,processed_data.iloc[:,-2],'k')
        ax.axhline(y=threshold,color='r',linestyle='-')
        ax.set_ylabel("Sum of Squares")
        ax.set_yscale("log")
        ax.set_xlim(mdates.date2num(starttime_dt),
        ↪  mdates.date2num(endtime_dt))
        ax.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M'))
    if show_norm:

        if show_power_filter:
            ax=axarr[2]
        else:
            ax=axarr[1]
        ax.cla()
        ax.plot(freqs[::skipch]*1e-6,median_data.iloc[-1,:],'k') #plot
        ↪  only last acquired median
        ax.set_xlabel("freq (MHz)")

    plt.pause(.3)



def main(argv):
    args=parser.parse_args(argv)
    metadata = get_metadata_from_h5(h5py.File(args.rawfile.replace('.r⌋
    ↪  aw','.h5')))

    processed_folder = args.processedfolder
    median_folder = args.medianfolder

    #Initalize list of filenames
    scaled_data_filenames = []
    median_data_filenames = []

    all_processed_data = None
    all_median_data = None

    fig,axarr=plt.subplots(1+args.show_normalization+args.show_power_f⌋
    ↪  ilter,1)
    while(True):
```

```
all_scaled_data_filenames = os.listdir(processed_folder)
all_median_data_filenames = os.listdir(median_folder)

new_scaled_data_filenames =
 ↪  sorted(list(set(all_scaled_data_filenames) -
 ↪  set(scaled_data_filenames)), key=natural_keys)
new_median_data_filenames =
 ↪  sorted(list(set(all_median_data_filenames) -
 ↪  set(median_data_filenames)), key=natural_keys)

scaled_data_filenames = all_scaled_data_filenames
median_data_filenames = all_median_data_filenames

#if new processed data is present
if(len(new_scaled_data_filenames) > 0):

    try:
        processed_filepaths = [processed_folder+"/"+filename
         ↪  for filename in new_scaled_data_filenames]
        new_processed_data = pd.concat(map(pd.read_csv,
         ↪  processed_filepaths), axis=0)

    except pd.errors.EmptyDataError:
        pass
    else:

        if(all_processed_data is not None):
            processed_data_list = [all_processed_data,
             ↪  new_processed_data]
        else:
            processed_data_list = [new_processed_data]

        all_processed_data = pd.concat(processed_data_list,
         ↪  axis=0)

        scaled_data_filenames = all_scaled_data_filenames


#if new median data is present
#Procedure is not needed for real-time plotting, as only the
 ↪  latest median is of interest. Still implemented for
 ↪  possible interactive plotting functionality later on.
if(len(new_median_data_filenames) > 0):

    try:
        median_filepaths = [median_folder+"/"+filename for
         ↪  filename in new_median_data_filenames]
        new_median_data = pd.concat(map(pd.read_csv,
         ↪  median_filepaths), axis=0)
```

```
            except pd.errors.EmptyDataError:
                pass
            else:

                if(all_median_data is not None):
                    median_data_list = [all_median_data,
                     ↪  new_median_data]
                else:
                    median_data_list = [new_median_data]

                all_median_data = pd.concat(median_data_list, axis=0)
                median_data_filenames = all_median_data_filenames

        if((len(new_median_data_filenames) > 0) |
         ↪  (len(new_scaled_data_filenames) > 0)):
            plot_real_time(fig,axarr,all_processed_data,
             ↪  metadata['freqs'],args.vmin,args.vmax,median_data=all_
             ↪  median_data,sampleSize=metadata[u'SAMPLING_TIME'],cmap
             ↪  =args.cmap,show_norm=args.show_normalization,
             ↪  show_power_filter=args.show_power_filter,
             ↪  threshold=args.threshold)


        time.sleep(args.wait_time)

if __name__ == '__main__':
    main(sys.argv[1:])
```

We can now start the experiment by issuing the following commands:

```
docker-compose up -d

docker exec -it spark-master /bin/bash

./spark/bin/spark-submit --master spark://spark-master:7077
 ↪  /opt/spark-apps/medianScalePowerFilterAstronBeamformed.py
```

followed by initializing the plotting procedure from a separate console:

```
python plot_median_scaled_data_with_power_filter.py -r
 ↪  L701913_SAP000_B000_S0_P000_bf.raw -p
 ↪  /mnt/spark-results/median_scaled_data -m
 ↪  /mnt/spark-results/medians -n -f
```

Finally the stream can be initalized by issuing this command from a third console:

```
python read_binary_buffer_write_csvstream.py -i
 ↪  L701913_SAP000_B000_S0_P000_bf.raw -d spark-master -fd
 ↪  /opt/spark-data/beamformed -fm /opt/spark-data/metadata -m 500 -w
 ↪  5 -t 10
```

A visualisation of 4 intermediate results of the online power filter figures can be found in Figure C.2.

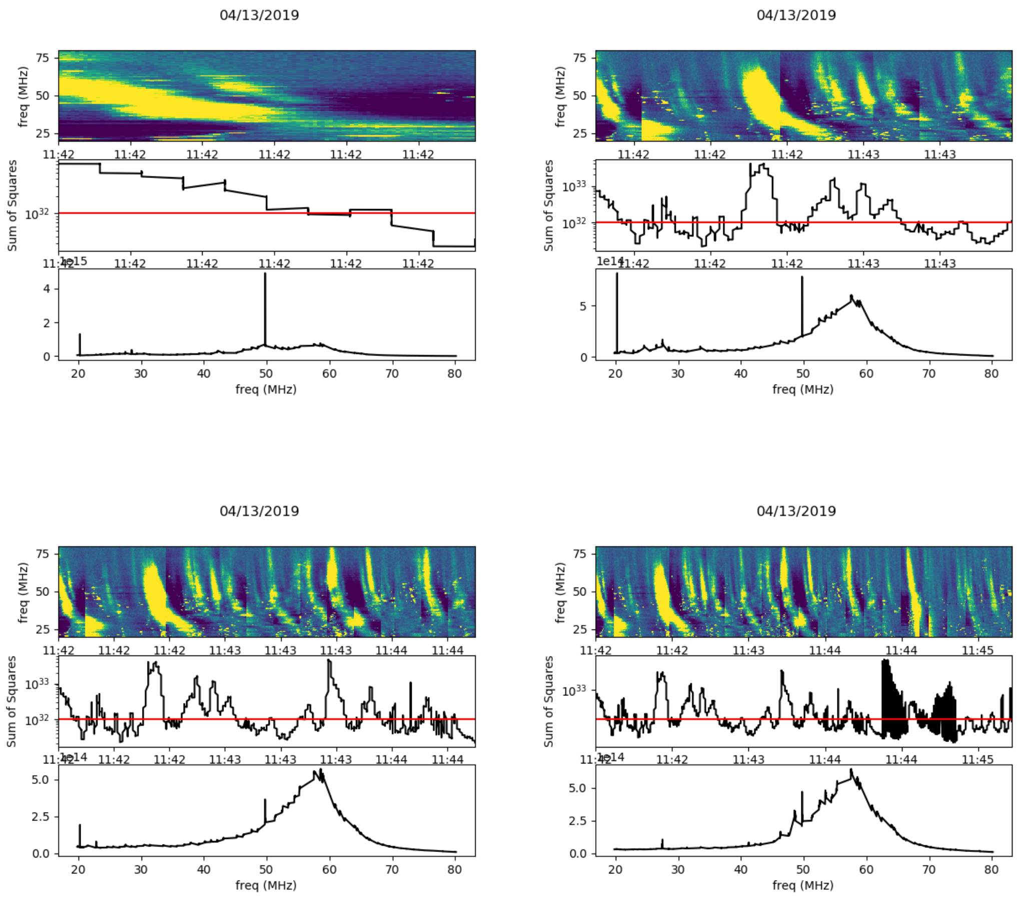

FIGURE C.2: Intermediate results from the online normalization and power filter calculation procedure. 4 intermediate results are shown. The upper part of each subfigure shows the normalized beam-formed, or dynamic spectrum, the middle part shows the power filter as well as an adjustable threshold, and the lower part shows the most recently calculated median, i.e. the one used to scale the most recent micro-batch. The power filter threshold is set to 1e32 in this example. In the dynamic spectrum dark blue indicates a low intensity, whilst yellow indicates a high intensity.

Bibliography

[1] DN Baker et al. “Effects of space weather on technology infrastructure”. In: *Space Weather* 2.2 (2004).

[2] DF Webb et al. “The solar sources of geoeffective structures”. In: *Washington DC American Geophysical Union Geophysical Monograph Series* 125 (2001), pp. 123–141.

[3] George Siscoe. “The space-weather enterprise: past, present, and future”. In: *Journal of Atmospheric and Solar-Terrestrial Physics* 62.14 (2000), pp. 1223–1232.

[4] Rainer Schwenn. “Space weather: The solar perspective”. In: *Living reviews in solar physics* 3.1 (2006), p. 2.

[5] JY Liu et al. “Ionospheric solar flare effects monitored by the ground-based GPS receivers: Theory and observation”. In: *Journal of Geophysical Research: Space Physics* 109.A1 (2004).

[6] John T Gosling. “The solar flare myth”. In: *Journal of Geophysical Research: Space Physics* 98.A11 (1993), pp. 18937–18949.

[7] Donald V Reames. “The two sources of solar energetic particles”. In: *Space Science Reviews* 175.1-4 (2013), pp. 53–92.

[8] SK Antiochos, CR DeVore, and JA Klimchuk. “A model for solar coronal mass ejections”. In: *The Astrophysical Journal* 510.1 (1999), p. 485.

[9] Space Studies Board, National Research Council, et al. *Severe space weather events: Understanding societal and economic impacts: A workshop report*. National Academies Press, 2009.

[10] N Balan et al. “A scheme for forecasting severe space weather”. In: *Journal of Geophysical Research: Space Physics* 122.3 (2017), pp. 2824–2835.

[11] HJ Singer, GR Heckman, and JW Hirman. “Space weather forecasting: A grand challenge”. In: *Washington DC American Geophysical Union Geophysical Monograph Series* 125 (2001), pp. 23–29.

[12] Rainer Schwenn et al. “The association of coronal mass ejections with their effects near the Earth”. In: *Annales Geophysicae*. Vol. 23. 3. 2005, pp. 1033–1059.

[13] Vasili V Lobzin et al. “Automatic recognition of type III solar radio bursts: automated radio burst identification system method and first observations”. In: *Space Weather* 7.4 (2009), pp. 1–12.

[14] MP van Haarlem et al. “LOFAR: The low-frequency array”. In: *Astronomy & astrophysics* 556 (2013), A2.

[15] P Chris Broekema et al. “Cobalt: A GPU-based correlator and beamformer for LOFAR”. In: *Astronomy and computing* 23 (2018), pp. 180–192.

[16] Hanna Rothkaehl et al. “LOFAR for Space Weather (LOFAR4SW) H2020 program”. In: *EGU General Assembly Conference Abstracts*. Vol. 20. 2018, p. 18974.

[17] Rik Jongerius et al. “An end-to-end computing model for the square kilometre array”. In: *Computer* 47.9 (2014), pp. 48–54.

[18] John W Romein et al. “Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer”. In: *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2006, pp. 59–66.

[19] John W Romein et al. “The LOFAR correlator: implementation and performance analysis”. In: *ACM Sigplan Notices*. Vol. 45. 5. ACM. 2010, pp. 169–178.

[20] Jan David Mol and John W Romein. “The LOFAR beam former: implementation and performance analysis”. In: *European Conference on Parallel Processing*. Springer. 2011, pp. 328–339.

[21] Jim Gray and Alexander S Szalay. “Where the rubber meets the sky: Bridging the gap between databases and science”. In: *arXiv preprint cs/0502011* (2005).

[22] J Bruno Morgado et al. “Very large scale high performance computing and instrument management for high availability systems through the use of virtualization at the Square Kilometre Array (SKA) telescope”. In: *Software and Cyberinfrastructure for Astronomy V*. Vol. 10707. International Society for Optics and Photonics. 2018, p. 107070I.

[23] Michael Armbrust et al. “Structured streaming: A declarative API for real-time applications in apache spark”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 601–613.

[24] Guido Rossum. “Python tutorial”. In: (1995).

[25] Matei Zaharia et al. “Spark: Cluster computing with working sets.” In: *HotCloud* 10.10-10 (2010), p. 95.

[26] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[27] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[28] Reynold S Xin et al. “Shark: SQL and rich analytics at scale”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 2013, pp. 13–24.

[29] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 15–28.

[30] Lei Gu and Huan Li. “Memory or time: Performance evaluation for iterative operation on hadoop and spark”. In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE. 2013, pp. 721–727.

[31] Wes McKinney et al. “pandas: a foundational Python library for data analysis and statistics”. In: *Python for High Performance and Scientific Computing* 14.9 (2011).

[32] Wes McKinney. “Pandas, Python Data Analysis Library”. In: *see http://pandas. pydata. org* (2015).

[33] Michael J Crawley. *The R book*. John Wiley & Sons, 2012.

[34] R Core Team et al. “R: A language and environment for statistical computing”. In: (2013).

[35] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1383–1394.

[36] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[37] Joseph Torres et al. *Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3*. 2018.

[38] Sanket Chintapalli et al. "Benchmarking Streaming Computation Engines at Yahoo!" In: *Tech. Rep.* (2015).

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

[40] Dag Sverre Seljebotn. "Fast numerical computations with Cython". In: *Proceedings of the 8th Python in Science Conference*. Vol. 37. 2009.

[41] Stefan Behnel et al. "Cython: The best of both worlds". In: *Computing in Science & Engineering* 13.2 (2011), pp. 31–39.

[42] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[43] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[44] Michael Greenwald and Sanjeev Khanna. "Space-efficient online computation of quantile summaries". In: *ACM SIGMOD Record* 30.2 (2001), pp. 58–66.

[45] Qiang Ma, Shanmugavelayutham Muthukrishnan, and Mark Sandler. "Frugal streaming for estimating quantiles". In: *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2013, pp. 77–96.

[46] Deepak Vohra. "Apache parquet". In: *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.

[47] George Heald et al. "Low Frequency Radio Astronomy and the LOFAR Observatory". In: *Springer International Publishing, doi* 10 (2018), pp. 978–3.