Universiteit van Amsterdam

Vrije Universiteit Amsterdam

Master thesis

Computer Science

Assessment of using big data software for remote sensing applications

Author:
Tom Peerdeman

Supervisor:
Adam Belloum (UvA)

February 28, 2017

Abstract

The "Downstream platform" is the name given to the infrastructure of software, tools and servers to develop services. These services use data generated by earth observing satellites. The current implementation of this platform is not very scalable. This research will asses the usage of big data software, to create a platform which can scale better with the large amount of data generated by the satellites. In order to do so an architecture will be designed and evaluated. To select the right software three processing platforms will be compared: Hadoop MapReduce 2, Apache Spark and Apache Flink. The comparison will be made by implementing an existing application for all three processing platforms, and evaluating the various aspects of those processing platforms.

Contents

Chapter 1

Introduction

Airbus Defence and Space Netherlands is a company located in Leiden that provides products and services for the international aerospace industry. The company works on the development and assembly of solar arrays, structures, instruments and other various systems for space. One of the projects currently being worked on at Airbus Defence and Space Netherlands is the Downstream platform. The "Downstream platform" is the name given to the infrastructure of software, tools and servers to develop and run a specific class of applications. These applications implement various services that provide access to aggregated data. This data is gathered by measuring devices present in earth orbiting satellites. This is also known as remote sensing or earth observation.

The applications that are currently implemented for the Downstream platform work fine as is, however, the platform itself was not really designed to cope with future expansions. It is expected that the earth observing instruments will improve over the years, this causes these instruments to generate much bigger data sets. This means that the applications need to be able to scale to handle the increase of data. Currently the platform and the applications have no or limited means of scaling. As the applications on the Downstream platform provide services, a fast response time is critical. If the data set size would increase too much, and due to limited scalability, thus the response time of this applications, the services would become useless.
A solution to this problem would be utilizing existing tools and software for big data, as they are built for scalability. In this research we will try to answer how and if we can utilize these big data platforms to enhance the Downstream platform. In order to do so we need to create an architecture based on these big data platforms. Some architectures already exists, as discussed in section 1.2.1. The problem with those architectures is that they can not be used as they use focus on different types of data or do not match in the scale of the data. Some of these platforms are also built around software which is now considered older. By creating an architecture from scratch, we can design a better one based on different software. To select the software that will be used, we shall compare the platforms based on the needs of the Downstream platform for Airbus Defence and Space Netherlands. To finalize the selection, the question how the software performs and if it scales has to be answered. Some benchmarks for big data processing platforms exists, as discussed in section 1.2.2. The problem with those benchmarks is that they where not created for remote sensing applications. The benchmarks focus on synthetic data and statistical analysis. To get an impression of the performance and scalability for remote sensing applications an existing application of the Downstream platform will

be used.

1.1 What is big data?

To be able to use software designed for big data we first need to know what big data actually is. A widely used definition is given by a big data study by the McKinsey Global Institute:
"Big data" refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze [1].

They acknowledge that the definition of "beyond the ability" can vary by sector, depending on what kinds of software tools are commonly available. This leaves us in the dark about the remote sensing sector. To check if we can speak of big data for the applications on the Downstream platform, we use the characteristics of big data. In a 2001 paper by Doug Laney three characteristics are discussed: *Volume, Velocity* and *Variety* [2]. These properties are still used today to characterize big data.

Volume refers to the amount of data. Big data is not called big without a reason. As mentioned before it is hard to define what amount of data qualifies as big. The amount ranges from terabytes to exabytes. Currently the Downstream platform houses applications that use input data that sum up to the range of terabytes. It is expected that future improvements to the remote sensing instruments will greatly increase this amount.

Velocity refers to the speed new data is generated. Besides the large volume of data already available, data in a big data environment is also generated very fast. This lead to the creation of streaming big data platforms. The difference between traditional batch and streaming systems is discussed in section 3. Data for the Downstream platform is also generated very fast. The earth observing satellites continuously generate new data.

Variety refers to the many forms the data can take. For example a dataset can be in the form of an unstructured text document. A different dataset can be a semi structured XML document. Big data can also be in the form of a completely structured database table. The applications of the Downstream platform all use some form of satellite data. This data is usually semi structured, and doesn't have much variety in that sense. The data however does differ quite for each earth observation instrument. Each instrument provides different data, and thus requires a completely different data structure.

We can see that these three characteristics indeed indicate that the remote sensing data from the Downstream platform qualifies as big data.

1.2 Related work

1.2.1 Related architectures

Most of the generated data by earth observing satellites are publicly available. Since most of these data sets are very large, it comes to no surprise that using big data software for exploiting such data is not a new idea.

An example of such an architecture is the PROBA-V mission exploration platform [3]. The architecture described is created and implemented to make use of the PROBA-V data archive. This data archive consist of measurements of the vegetation for climate impact assessment, surface water resource management, agricultural monitoring, and food security purposes. Besides this data the mission exploration platform also contains the SPOT vegetation data and is planned to also contain the data for Landsat 7 and 8, and Sentinel 2 and 3. The goal is to allow the user to access the data, an derive products from it to increase the use of the data. Any of this data or products can be accessed via a web portal as well as standardized discovery, viewing and data access interfaces.

The architecture of the PROBA-V mission exploration platform is based on Hadoop. The data is stored on either a shared storage, which is accessible via the network, or via the Hadoop distributed file system.
The processing is done using Hadoop MapReduce and Spark. A work flow engine called Oozie is used. This system allows the data to be processed as if it were a work flow, and starts the MapReduce and Spark processing platforms to do the actual processing in the flow as needed.

The next project is the German data access and exploitation infrastructure [4]. The system was built for the Copernicus data. This is also one of the data sets of which the Downstream platform is aimed at. The architecture features a portal to access the data, and its processed products. This system is similar to the PROBA-V mission exploration platform in that it is also based on the Hadoop software. The platform makes use of the Calvalus environment [5]. This environment uses the Hadoop distributed file system to store the data and provide data locality. The processing itself is done via Hadoop MapReduce. Note that this environment was introduced around 2009. This means that at the time no other mature systems other than the Hadoop software existed. This leads to the environment being built around MapReduce. Today many more processing systems exists and are being developed. This allows us to compare those systems, and see if MapReduce is still a sensible choice.

A project different from the PROBA-V mission exploration platform and the German data access and exploitation infrastructure is the processing of Euclid data. The Euclid mission is a survey mission developed in order to study the Dark Energy and the Dark Matter [6, 7]. This mission results in about 175 PB of data to be processed. While this processing is technically not processing of earth observation data, the processing of patches of sky can be related to processing patches of earth. The architecture can be related to the PROBA-V and German system, but is not based on Hadoop. The Euclid system also differs hugely in scale. Where the PROBA-V and German system are built to scale to multiple servers, the Euclid architecture scales to the magnitude of data centers.
The storage of the Euclid system is handled by a custom storage solution. The data is distributed over the various data centers. To access this system, the location of a data item has to be looked up. The locations of the data are stored in a central meta data repository.
The processing is again a custom system which is based on the map reduce model. It acts just like the Hadoop MapReduce system just as the PROBA-V and German system use. The difference between those systems is that the Euclid mission operates on a much

larger scale. The first step is to split up the data on a very large scale. The measured sky is split up in patches. Each patch is assigned to a data center, which is now responsible for the full process of processing that patch. The data centers itself can then split up those patches up to single observations in order to process them.

1.2.2 Comparing big data processing platforms

The second part of the research is to compare the processing platforms to pick the best one. One of the criteria are performance and scalability.

A rough comparison between the performance of various processing platforms can be given by the Daytona GraySort challenge. The goal of this benchmark is to measure the performance of a whole system by sorting large amounts of randomly generated data [8]. The minimum amount of data to be sorted is 100 TB. The performance is measured in TB per minute sorted. When looking at the winners in the result we can see the big data processing platforms Apache Spark [9] and Hadoop MapReduce [10]. The MapReduce implementation won in 2013 with 1.42 TB of data per minute sorted. The Spark submission beat this result in 2014 with 4.27 TB/min. This would indicate that the Apache Spark processing platform does perform much better. Comparing these results is difficult however. The hardware used to achieve those results differs in many ways. The MapReduce submission uses a lot of servers, while the Spark submission uses way less servers. The Spark submission uses more cores per server, more memory per server and SSD's. Also we have to keep in mind that the development of the platforms is still very much active. This means that the MapReduce implementation could be a lot faster a year later, when the Spark submission was made, than when the MapReduce submission was made.

The Big Data Benchmark by the AMPLab of the university Berkeley is a performance benchmark where the hardware is kept the same [11]. This benchmark measures the performance of running queries on data warehouse solutions. The Hive data warehouse uses Hadoop MapReduce to execute these queries. The Shark warehouse is built on top of Spark. While the processing platforms are not directly tested themselves, the queries that are being ran to be tested are eventually executed by the processing platforms. This thus allows for a comparison between Spark and MapReduce.
The results of this benchmark show that the Shark warehouse beats the Hive system by a lot for each query ran. These results of course include the overhead of Shark and Hive over the raw Spark and MapReduce performance. It may be clear though that the Spark system is much faster than the MapReduce system.

A benchmark that measures performance which does use the processing platforms itself is BigDataBench [12]. This benchmark consist of 19 applications. These applications work on data that is structured, semi-structured and unstructured. This means that the benchmark does contain applications that run queries on data using Hive and Shark, just like the Berkeley benchmark did. Other applications are directly implemented in Hadoop MapReduce and Spark. This allows for a direct comparison in performance for those systems.
The data itself that is used for testing is synthetic. It is generated using a tool called Big Data Generator Suite [13]. This tool generates the various synthetic data sets from real collected data sets.

Skipping forward to the description of the architecture in section 3.2.2 we see that besides Spark and MapReduce we will also make use of Apache Flink as one of the candidates for the processing platform. This platform is much younger, and therefore not yet considered in the previous mentioned benchmarks. A more recent paper called *Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks* compares Spark and Flink in terms of performance and scaling [14]. It does so by running 6 different batch based applications which cover the various aspects of the processing platforms. The results of the paper show that Apache Flink is faster in most applications. Only the grep application shows a result where Apache Spark performs slightly better.

Chapter 2

User needs

To be able to define a proper architecture that makes use of existing big data platforms, we first have to define what are the important aspects of this architecture. Using these aspects we can select the appropriate software for the architecture that fit the best.

2.1 Applications

The Downstream platform houses a wide variety of applications. The obvious difference between the applications is the processing steps taken. A difference that is just as important for the architecture is the amount of data that has to be processed. For example some applications use data sets with a resolution of 25 by 25 kilometer, however there are also applications that use 100 by 100 meter as resolution. The latter application will have a much bigger data set to process. The architecture should be generalized to cope with such differences.
Since the many applications run on the same architecture, it is likely that the applications will be ran at the same time. The architecture should therefore support multiple applications being able to run independently (not counting inter application dependencies) from each other.

The applications implement services, this means that the end product is created on the fly as the user requests it. Some processing can be done via offline processing. The term offline processing refers to processing of the data via automation, that is the end user did not explicitly ask for the result. This offline processing could be the processing of end results, as this for example might speed up on demand processing. A different form of offline processing is pre-processing of the whole data set. An example of such pre-processing is filtering, interpolating and combining the various data sets. The online processing can then use the offline results to generate their result.
These various data sets might not be just data generated by satellites. For some applications it is feasible to use and process additional data sources apart from the remote sensing data. An example of such data is ground data. This ground data can be used, for example, as means to verify satellite measurements. The ground data can also be used to generate additional insights on top of the satellite data.

2.2 Creating new applications

Besides the aspects of running the applications, creating new applications is also very important. In order to do so it is important for the developers that the programming is a simple as it can be. This is mainly influenced by the programming API's and programming model of the processing and storage platform. Another factor is the programming language used. Most of the existing applications are written in Python, as the main programming language used at Airbus Defence and Space is Python. It is therefore useful for the application developers that the architecture will support applications written in Python.

2.3 Administration

The platform can be used by multiple users. This means that multiple users can request on demand processing at the same time. The architecture should therefore support running multiple applications, and multiple processing jobs of the same application next to each other.
Not every user should be able to start a processing job though. Through the accounting interface the administrator can modify what end products can be accessed by each user. The usage of those products by the individual end users should also be able to be monitored via this accounting module.

2.4 Security

Since the products are only available to certain users, it is important that the accounts are only accessible via a strong authentication mechanism. If the user account were to be compromised unauthorized access to the products could occur, or wrongful charges of usage be applied.
Some products or data sources might need stronger protection from unauthorized access. One of the methods of protection is encryption. This means that input data sets are possibly encrypted, and need decryption before they can be used. It is also possible that the intermediate data sets also need encryption and decryption. A second enhancement for stronger protection is the separation of the data. This means that not all applications can access all data sources or intermediate products. Instead the architecture should separate these data sets and only allow access to the applications that need it.

A different security measure is required for the applications. Some of the applications generating the end products are closed source. Therefore the architecture should hide the actual application logic from the end users. The code or compiled versions of the code should not be accessible by the end users. However as the applications are created as services, this aspect will very likely be fulfilled, as services only return results. When using a service the end user never sees an executable, or any form of the application code.

2.5 Performance

One of the goals of using a big data platform for the Downstream platform is to improve the processing time by utilizing more nodes. It is therefore expected that the architecture delivers a reasonable processing time. The expected upper limit of the processing time of course depends on the application. The processing time itself depends on the scale of the cluster running the architecture. To give an indication of an expected processing time: The on demand processing should return a result in terms of seconds, as it is a service, and they require short latency's. The processing time of the pre-processing is less important as it should only be done once. It is expected that new data arrives with intervals of hours. The maximum processing time should therefore also be in terms of hours.

As multiple users are using the system, it is likely that multiple users will use the system at the same time. It is however unwanted that the performance, and thus the processing time will suffer from the multiple users. The architecture should therefore be designed to allow multiple jobs running simultaneously without impacting the performance, and thus the processing time, too much.

2.6 Scalability

Due to increased data size, more users or even more applications more processing power may be required. The architecture should therefore be designed to scale in the amount of processing power. The preferred scaling factor is linear, meaning with n times more nodes, the processing time decreases by $\frac{1}{c} \times n$ times, where c is a constant greater than 0. The perfect linear scaling would be a situation where c equals 1.

Due to this increased data size, the storage capability of the system also needs to increase. The architecture should therefore be able to scale its storage. The scaling can be done be either scaling up via adding more storage to each node, or via scaling out by adding more nodes. Initially the system will store hundreds of terabytes of data.

2.7 Portability/Mobility

After the creation of the architecture it may be, due to various reasons, that the chosen big data platform has to be replaced with a different platform. Therefore it is important that the impact of the chosen platform is minimal to the application. This means that the applications can easily be ported to a different platform if needed. It is also possible that this new platform is deployed next to the existing one. In such a case the platforms should not interfere with the workings of each other.

It may be beneficial to Airbus Defence and Space to move the applications to an existing other cluster, or even the cloud. Also due to scaling the cluster may grow or shrink in number of nodes. The architecture should therefore be easily movable between hardware configurations. This means that the applications can not have hard coded server addresses, paths or even requirements for the amount of nodes.

2.8 Reliability

Since copying hundreds of terabytes of data, or worse, losing irreplaceable data is a horrible situation, a form of data reliability is needed. The architecture should support some form of reliability, which allows the data to be recovered in case of partial data failure.

The same principle should apply for the processing. If an processing node happens to fail, and become unresponsive, the processing system as a whole should continue to work. If possible it would be beneficial if the system could detect the failure and redo any work that had been lost. This allows for execution jobs to still finish, even when node failure occurs, instead of failing the execution due to lost processing work.

2.9 Cost

The final important aspect for the architecture is the cost in terms of money of running it. It makes sense that if the architecture is perfect except for the cost of running it being very high, it won't be used. The running cost includes any licences and hardware requirements. The goal is to design the architecture in a way too keep the running cost as low as possible.

Chapter 3

Architecture

3.1 Identifying the components of the architecture

Now that we have identified the important aspects of this architecture, we can start shaping the architecture. The first step is to identify the components that are required. Let us start with the processing. In section 2.1 we identified that some applications may need some form of pre-processing besides the on demand processing. We can thus already identify two important components of the architecture: The pre-processing and the on demand processing.
The pre-processing step generates intermediate results, that can later be used by the on demand processing. But what happens with these intermediate results in the meanwhile? It is expected that due to the size these intermediate results do not fit in memory. Note that this hypothesis is tested in section 5.4. The alternate solution is to store these results in an intermediate storage. The intermediate storage component stores the result on disk, which is usually much larger than the memory.

The next component is also storage related. The architecture needs a component that allows the storage of the input data. As the input data is likely to span multiple Terabytes, a simple single (shared) disk solution will not work.

The final component that we can identify is the interface to the user. The interface should be the only means that any user can interact with the system. This includes the managing of the users, that is any accounting is also done via the interface.
In figure 3.1 we can see the interaction between the components when a regular user requests a new product on demand. We can also see the interactions when new data arrives, and the pre-processing is started.

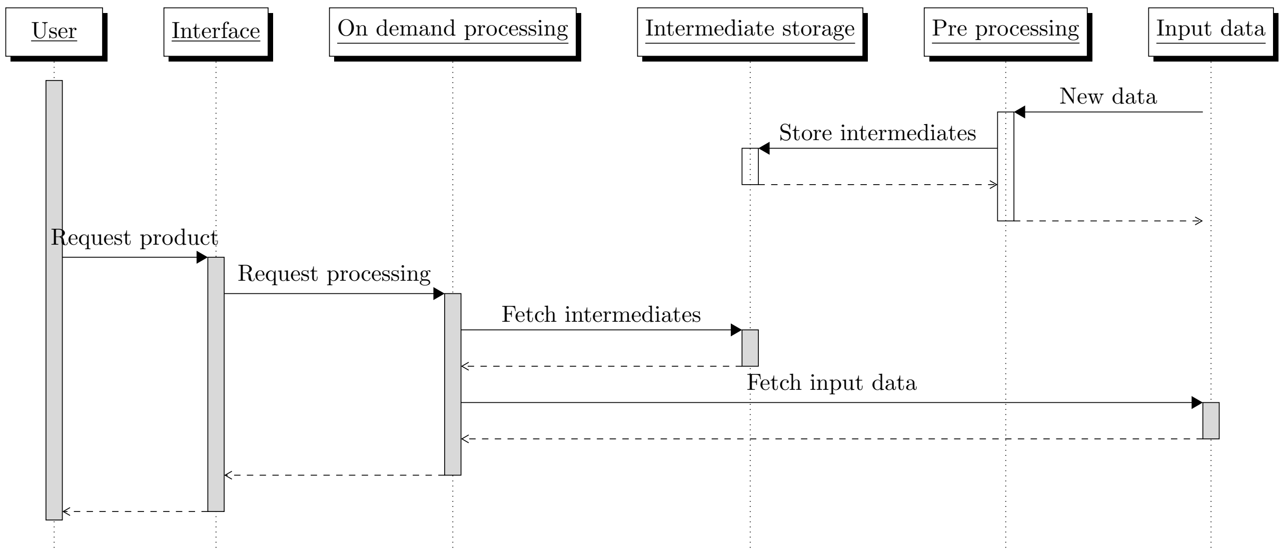


Figure 3.1: Sequence diagram of interactions between the components of the architecture

From this diagram we can also spot an interesting problem. The point in time where the on demand processing and the pre-processing are started are independent. This means that they also could overlap. This can create a situation where the on demand processing is reading intermediate data while the pre-processing is in the process of writing to that same intermediate storage, and could be writing the exact same dataset the on demand processing is currently using. This could lead to wrong results, or even crashes, as the on demand processing might use a partition of new data, while the pre-processing did not finish the complete updated data set.

3.2 Software

Now that we have identified the components, we can start fulfilling these components by selecting software. In figure 3.2 we can see that the architecture is split into three parts. The application layer represents the interface and the application code. The actual contents of this layer is application specific. Each application of course requires different application code, but it may also require a different interface.
The processing layer contains most of the components we identified. It contains the software for storage and processing. This layer is shared by all applications.
The final layer is the infrastructure layer. The infrastructure contains the hardware which is used by the processing layer.

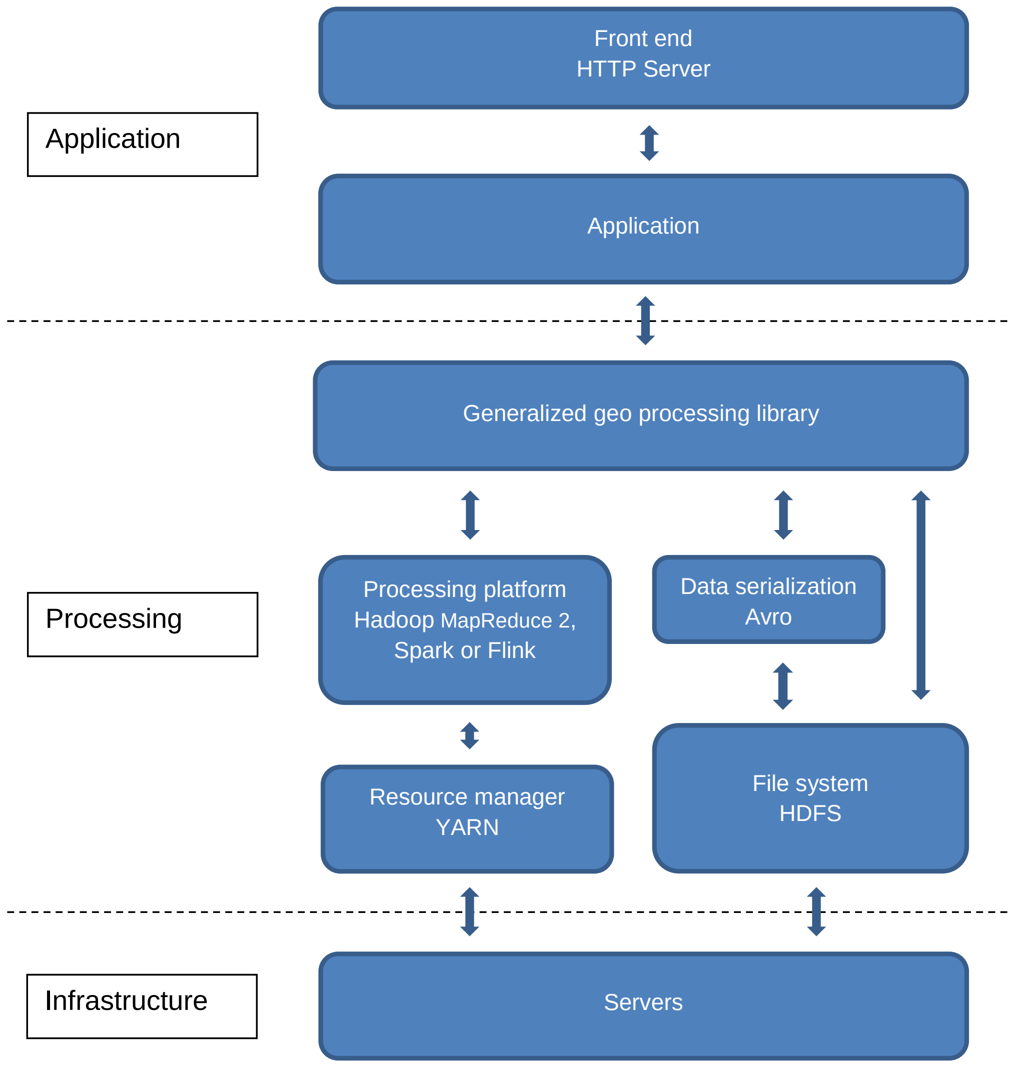


Figure 3.2: Diagram of the software of the proposed architecture

3.2.1 Interface

The interface is the bridge between the user and the processing system. The interface can, based on requests, start new processing jobs. The interface also serves as means for accounting. It should therefore measure the usage of the users. It should also deny any access if the user has no permissions. As mentioned before the actual interface is application dependent, however some general statements can be made. Since the user interface has to reach a broad spectrum of clients, an internet, via HTTP, based interface would be the easiest to use, as it allows for simple access. Every user which has access to the internet is very likely to have a browser installed. Any visual product, being graphs or pictures, can thus be shown in the form of web pages. No additional software is required for the clients. Results that consist of numerical data are often used by applications. Web based services that provide such data often use a REST API, which is based upon HTTP. This allows for easy building of the applications, as libraries for interactions with REST API's already exist. Such HTTP based interfaces however rely on the fact that the on demand processing should complete and return a result within seconds. If a HTTP requests takes more than a couple of seconds usually the browser issuing it will time out, and show an error message to the user.

The interface only exists to pass along the end results, it does not do any heavy processing on large data sets. Therefore the scalability of the software is not too important. The interface can therefore be implemented by a very simple web server backed by a (relational) database. In this database the users with their login credentials and permissions are stored. This database can also contain the usage of each user.

3.2.2 Processing platform

Let us start with the main software component of the architecture, the processing platform. This platform will execute the functions of the pre-processing and the on demand processing. To determine the platform we first have to know the difference between streaming processing platforms and batch platforms. Streaming systems assume that the data provided as input is infinite, whereas batch systems require finite data. Processing batch data is thus known to terminate, while processing streaming data may not terminate at all. These systems also have slightly different performance metrics. Batch system are focused on throughput, how long will it take to process all the data in the batch? For streaming systems this metric is also important, however latency is even more important. Applications utilizing streaming platforms usually operate on live data. It is thus important for the application to have the data as fast as possible processed starting from the time the data was created.

This architecture focuses on applications which are batch based. While the input data is constantly generated, and thus looks to be an infinite stream, new data that is generated is delivered in big batches. The applications generate results from the fixed amount of data that is available at the time of execution. An application that would use an infinite stream would constantly generate results and never stop doing so, which none of the applications used do. The processing is thus applied on batches of data, and not on an infinite stream. Therefore we can ignore all the stream processing platforms that are available for this architecture.

The basics ideas of big data batch processing originate from the map reduce model, created in 2004. The map reduce model describes a model to analyze large amounts of data by splitting it up in to key value pairs. The actual processing in the map reduce model is done in two simple steps: map and reduce. In the map stage the key/value pair is processed to produce (multiple) intermediate key/value pairs. In the reduce step these intermediate pairs are merged to create the output. The strength of this model lies in the fact that the map stage can be spread over many machines, same with reduce, thus creating a very scalable system.

Among the various implementations, one of the most widely used implementations of this model is called Apache Hadoop. Hadoop is an open source project and is, after years, still being actively developed. It could therefore be a very good candidate for our architecture. Since the project has no dependencies on legacy versions, it would be wise to use the latest stable version. The latest recommended version of Hadoop is, at the time of writing, 2.7.2 [15]. Note that releases in the 2.6.x line are also available The 2.6 line is maintained due to major API changes in the 2.7 line. It would therefore not be wise to use the 2.6 API, which may become obsolete at some time in the future. A 3.x line is also available, however these releases are still in alpha phase. Since we want a stable system it would not be wise to use the alpha releases.

Of course the developments of big data processing platforms has not stopped in the years since the map reduce model was introduced. In 2010 a project now called Apache Spark was made open source. Spark is claimed to be the next generation as it could outperform Hadoop by utilizing in memory computation. It is a project that is slowly reaching maturity, and is being picked up by the industry. The programming model of spark is a bit more extensive than the map reduce model. Spark still has the map and reduce functions which operate on key/value pairs. However Spark also supports a range of other functions like joining data, filtering data, and much more other transformations. It can do so on scalar data and tuples with two or more values. This makes this platform very interesting for the architecture. The version used is Spark 2.0.0, which was released July 2016. The 2.x releases come with a bunch of improvements, including performance, over the 1.x releases [16].

A similar system is called Apache Flink. Flink uses a programming model similar to Spark in that it knows the same transformations and also can use tuple data. Flink is different from Spark as it is a streaming platform. For the architecture streaming is not very useful. However Flink was designed very well, and can process batch jobs as well. It does so by treating the batch data as if it were a stream. These streams are however special, since normal streams are infinite.

The nice thing about these systems is that they are very interchangeable. Both Spark and Flink can directly use the input and output formats to and from disk that Hadoop uses. Hadoop comes with a resource management system to schedule jobs, called YARN. Both Spark and Flink support this system. This and the fact that the programming models are fairly similar make these platforms quite interchangeable, which is a requirement of the architecture. The latest stable version of Flink is 1.1.0 [17].

Besides these three systems other systems do exists. For example many other implementations of the map reduce model exists. However these systems have not broken through in popularity like Hadoop, Spark and Flink. This leads to questions if the support is good enough, and if the platform is stable enough. Other systems not based on map reduce also exist, for example GraphLab and HPCC [18]. The problem with such system is that they have a very different programming model that does not match the map reduce model. This leads to very poor portability. Due to this, and the fact that a lot of systems do not support python as programming language these platforms are rejected as candidates for the architecture.

This leads us to three candidates as platform for the processing. To test these platforms better a example application is implemented. After this implementation is done an evaluation can be made to determine the best big data processing platform for this architecture.

3.2.3 Intermediate storage

The important aspects of the intermediate storage are scalability and performance. Since it is very likely that the intermediate storage will contain many terabytes of data, it has to be scalable in the amount of storage. When this storage has to feed the many parallel processes that are executing on-demand jobs, it is also very important that the system can scale. A central storage solution will not work very well. A single server isnt very likely to perform well while reading multiple data pieces at the same time, while sending them to the appropriate processing nodes.

Let us look at the kind of data the storage will actually hold. Since most of the input data is based around satellite data we can expect the input data to have coordinates, a measurement and other meta data. The pre-processing will likely filter, and combine these measurements, resulting in data types for coordinates, measurements and meta data. This structured data format closely resembles a row in a table. This leads to the possibility of using traditional relational databases for the architecture. The problem with these databases is that they do not really scale very well. Traditional databases often scale by mirroring or splitting up the data. This can lead to performance issues due to maintaining consistency and row locking overheads [19, 20]. These kinds of problems lead to the increase of popularity of NoSQL databases. A NoSQL database is related to the traditional databases, but can drop some features like the relations between tables, atomicity, a fixed data schema, and more. This allows them to scale better. Since the data does not have relations between tables, and the architecture does not need those extra features, NoSQL databases seem to be a good fit for the architecture.

If we look back at the requirements we see that this solution does not fit perfectly. If we look at the performance there is a slight issue. Let us imagine we have used a NoSQL database, for example Cassandra. The data stored in this database is spread out over all the nodes, as the database scales. If we now start a new processing job, running on those same machines, we would like the process on each node to start processing the data that is local to that server. Using a NoSQL database the job scheduler has no idea where the data actually is located. Thus this step to improve data locality, and thus performance is not possible.
The second requirement that can not be fulfilled is security. Only a few NoSQL database implement multi tenancy. That means that multiple users can use the database, but their data is inaccessible to other users. Encryption of the data is only available in a very few of the available databases. This leads to a different solution which is not based on a database.

The first step of this solution is finding a piece of software that can store our data in a reliable way, but also be scalable. This software is already present in the form of HDFS. HDFS, Hadoop distributed file system, is a part of the Hadoop ecosystem. It facilitates distributes storage by splitting up files into blocks and spreading those blocks over the nodes. It can achieve reliability by creating redundant copies of all the blocks. It can meet the security requirements as it has built in support for data separation by access control and possibilities for encryption [21, 22].
While HDFS can run independent from MapReduce, MapReduce does require HDFS. Hadoop MapReduce is built on top of HDFS, and uses it by default as temporary intermediate storage. This means that it has built in support for the reading and writing to

and from HDFS. It also allows Hadoop to directly query the location of the data, and thus schedule accordingly. The means of translating in memory objects to storage, in this case HDFS, is called serialization. MapReduce requires that each implementation of a key or value object has a method to write itself to, and read a new object from a byte stream. In this way a long list of serialized objects can be generated by concatenating the bytes generated by each write call of all the objects. This list of serialized objects plus some extra meta data is called the sequence file format.
This support can also be used in Spark and Flink. However while Hadoop only uses key/value pairs, Spark and Flink can (and will) also use other data types. For example in the air quality application we have records of latitude, longitude, time, measurement and other meta data. A tuple of 5 or more values would be a better fit instead of a key value pair. Since the sequence file format only supports key/value pair reading and writing, this format cannot be used. Instead a different solution has to be found to translate the software representation of the data into a format that can be stored in HDFS.

Quite a few serialization libraries exist. However not all of them are suitable. Some of them do not have Python bindings, and can thus not be used. The first decision that has to be made is row oriented storage versus column oriented storage. Row oriented storage stores each row, which is a combination of latitude, longitude, measurement, meta data, in a long list. Column oriented storage stores each column in a list. Thus the latitude of each row is stored together, next follows a list of longitudes, etc. The benefit of column oriented storage is that if only a certain column is required, the data access can be very fast [23]. In the case of this architecture this will not happen very often. The on demand processing, who reads from the intermediate storage, often needs more than one column, and very likely the whole row.

Three of the popular row oriented serialization systems are Apache Avro, Google Protobuf and Thrift. Performance wise Protobuf is the fastest, followed by Thrift and finally Avro [24]. The benefit of Avro is that is uses a predetermined schema to indicate the structure of the data. While Protobuf and Thrift also have such a schema to define the data structure, it is only used when writing the data. This means that when reading the data the structure has to be extracted from the stored serialized data. This processes needs additional data to be available in the serialized data, and thus extra storage capacity [25]. This means that very often the final serialized data size is smaller when using Avro than Protobuf or Thrift [26, 27]. For this reason Avro is chosen as serialization system for the architecture. The version used is the latest stable version, which is 1.8.1 [28].

When we identified the components in section 3.1, we noticed a potential race condition in figure 3.1. It is possible that the pre-processing of an application is running while the on demand processing is called at the same time. This may lead to incorrect results of the on demand processing. A very simple solution to this problem is writing the pre-processing output to a temporary location in HDFS. When the pre-processing finishes, its final step is to move the output data to the correct location, where the on-demand processing expects new pre-processed data to be. This move is very cheap in terms of performance due to the set up of HDFS. A file is simply a string location which points to a block of data, which is distributed over one of the nodes. When moving this location string is altered, but the actual data can stay exactly where it was.

3.2.4 Input data

While the actual input data is highly dependent on the application using the data, we can determine how this data is stored. As said before HDFS provides reliability, security and possibly an performance increase due to locality. It would therefore be logically to also use this system for the input data.

The data of the air quality application is usually stored in HDF5 format [29]. This format uses a hierarchy to store the various data components that the data sets may provide. An implementation for reading HDF5 files directly into systems like Hadoop is not available. However Python has as package called tables that can read them. Various HDF5 libraries are also available for Java and Scala. A wrapper can therefore be written to get the data into the big data processing platform. Other applications may use different data formats, these formats can be loaded into the processing systems via the same process, via a library wrapper.

3.2.5 Generalized geo processing library

It is very likely that the people who create and maintain the applications for the Downstream platform are not big data experts. The application developers do not know the programming models of the processing frameworks, or how they work. They do not know Avro, and do not know how to use HDFS. To help these application developers an additional component should be added to the architecture. A programming library is needed that can abstract the various architecture details away, and simplify creating applications that utilize satellite data. This library should specialized to satellite data based applications, but also be generalized for those applications, so it can cater to the various needs of the range of applications. This library should handle the mapping of application code to the chosen processing platform, simplify the intermediate data storage, and integrate the accounting into applications.
A platform that abstracts the processing is already available, and is called Apache Beam. Apache Beam provides an API that allows the user to provide a pipeline of transformations. This pipeline is then executed by a runner, which maps this pipeline on the processing platform for that runner. At the moment of writing a runner for both Spark and Flink exists. The first problem with Beam is that no MapReduce 2 runner exists. This is because the pipeline model does not map easily to the MapReduce model. The second problem is that this pipeline model is not specialized in satellite data, and closely resembled the Spark and Flink API, which we tried to abstract away. In this way Beam does not abstract away the programming model, it just unifies the programming API's of the various processing platforms. To create the full geo processing library, the remaining solution is a custom library that can handle all the requirements. Apache Beam could be used as a layer below this library, to mask the actual processing platform used, and thus allow for a simpler implementation of the custom library. The actual hiding of the processing model, to allow for easier application development, has to be implemented in this custom library itself.

3.3 Hardware

An aspect of the architecture just as important as the software is the hardware requirement. The actual number of servers, known as nodes in a cluster, and the composition of these servers depend on the requirements of the application [30]. For example if the application is compute intensive it would benefit more from more CPU cores. If the application does very little processing, it makes sense to speed up the disk I/O by using more or faster disks.
In the case of the Downstream platform it is still unknown which application types are going to be ran. It would therefore make sense to focus on a general machine type which is not focuses on either compute or disk I/O alone.

3.3.1 Types of the nodes in the cluster

Assuming we have multiple machines to use, the arrangement of them is very simple. Map reduce platforms use a simple master slave structure to control the cluster. One machine is more powerful than the others. In older versions of Hadoop MapReduce this machine becomes the master. This node accepts jobs and controls the execution of that, and all the jobs. All the other nodes are slaves, also known as workers. These workers execute code as instructed by the master.
The problem with this structure is that one node has to track all the jobs sent to it. This can be quite a lot of work, an thus does not scale very well. Hence YARN, yet another resource negotiator, was created. YARN replaces the single master node with a resource manager node. The jobs still arrive at this resource manager node. The actual job master, which tracks the status of the jobs, is then delegated to a slave by the resource manager. This slave thus becomes a temporary master node, just for that one job.

When HDFS is used, an entry point is needed. This is called the namenode. This namenode keeps the directory structure of the whole distributed file system in memory. The actual data is stored in datanodes. A logical mapping would be a datanode on each slave. That way the system can schedule jobs so that most of the data needed for a particular task is already present in the same node. The namenode can be ran on either the resource manager node, or a separate machine. An example overview of a deployment of nodes can be seen in figure 3.3. In this example 3 slave machines exists, the resource manager and namenode run on the same machine. Note that the namenode and the resource manager can have standby instances that immediately can take over if the original fails. If such high availability is required, another identical resource manager machine is needed. The same counts for the namenode, if it does not share a machine with the resource manager.

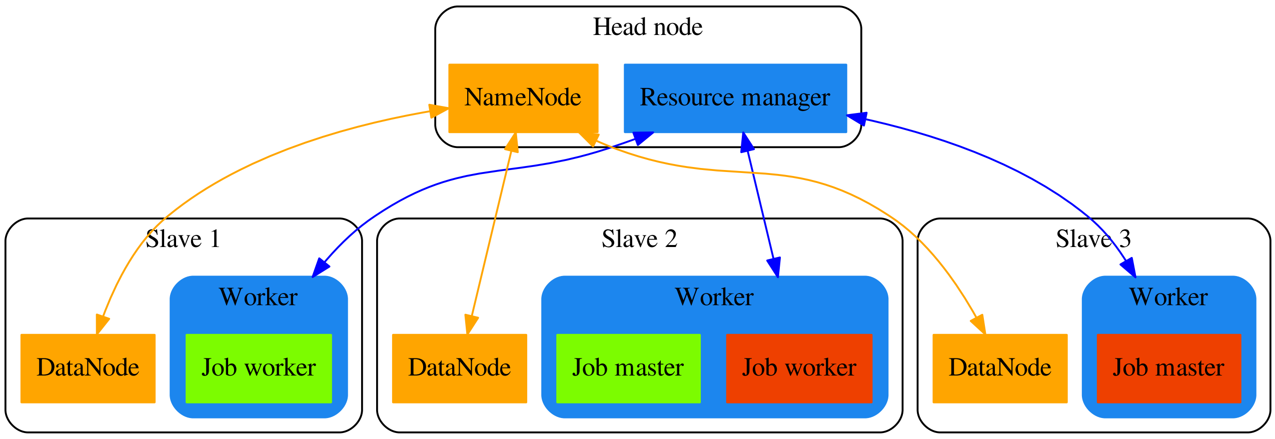


Figure 3.3: Example node overview of a 3 slave cluster running 2 jobs

3.3.2 CPU scaling

When the power required for processing your application surpasses the power you have available you have to scale your existing solution. This process of scaling can be done using two methods: scaling up and scaling out. The scaling up method consists of acquiring more powerful hardware to replace your existing hardware. This usually means getting a CPU with more cores or having a higher clock speed. The method of scaling out means keeping your old hardware, and adding new hardware. This new hardware does not have to be better than the old hardware.
This leaves us with an interesting question for the architecture: Should one get a better CPU with more cores, a faster CPU, or more machines with a less impressive CPU? The processing platforms used in this architecture make use of YARN, which allocates virtual cores and memory to an application. Each virtual core usually represents one physical core. The allocation of virtual cores simply represent a percentage of the physical CPU's processing power. The execution of code is handled by the operating system, which guaranties no core affinity. This means that for example an application with a one vCPU allocation running on a quad core, may at one point run on core 1, and a second later on core 3.
The mapping of tasks to virtual cores depends on the processing platform used. However task always requires at least a single virtual core. Therefore more machines, and a CPU with more cores, will give more task slots. This leads to faster processing, as more processing can be done at the same time. However faster cores also lead to better performance, as each task requires less time to complete. In a lot of applications there will come a point where the data can not be processed parallel anymore, and has to be redistributed. Generally the reduce part of map reduce require a redistribution of data. One of the slower parts of a computer is still the network. It would therefore make more sense to use workers with more cores, as some of the data, if not all, then does not need to be pushed to different machines, but can be redistributed in between the tasks in the same machine. However more cores or faster CPU's come at a cost. It may be cheaper to get two cheaper machines than one high end one, while the two may outperform the high end one together. It is therefore a balance between the CPU power per machine,

and the number of machines.

3.3.3 Storage

The requirements on the size of the storage are very simple. HDFS splits every file into fixed size blocks. These blocks are then distributed over the datanodes. Each block is also replicated on different datanodes for reliability and performance reasons. The default replication factor, which is configurable, is three times. This means that the actual storage requirements are at least three times the expected data size. The storage capacity on the namenode also needs to be of reasonable size. The namenode stores the whole file system tree, which holds the information which file and blocks are located on which datanode. This tree is also stored on disk, in case the one in memory one gets lost due to unexpected stops of the namenode process.

Chapter 4

Implementation of the air quality application

The processing platforms where chosen based on some of the important aspects we defined in chapter 2. Some important aspects were not mentioned when choosing though. For example scalability and performance are barely mentioned. We still want to evaluate our architecture on these aspects. The problem with these aspects is that they are qualitative. Developers can claim that their processing platform is faster than the other, but they can't give an exact number, as it differs for each application and implementation. Another aspect is the ease of implementing new applications. While the documentation may claim the the programming API is very simple, only trying it can prove that. To complete our evaluation of our architecture we therefore have to implement an proof of concept application. This way we can also evaluate the performance and scalability. As we have three version of the architecture, using MapReduce 2, Spark and Flink, we can use the performance and scalability to help us decide which processing platform is the best suited for this architecture.

Note that as this application is only a proof of concept not the whole architecture was implemented. This was done because the goal of this research was to assess the usage of big data tools for the Downstream platform. The goal is not to deliver a running full version of the architecture that can be immediately used for business means. This means that less interesting parts such as the front end, including administration are not created nor used. Instead the applications are directly started via the command line, and the results are manually retrieved.
Another important omission is the absence of the generalized satellite data processing library, which allows for easy programming of satellite data based applications on top of the components of the architecture. Implementing such a library would require at least lots of application domain knowledge, precise requirements, integration tests and lots of time to implement. While such a project is very interested as future research, such resources are not available for this project

4.1 The air quality application

The application we will be using is the air quality application. This application is an existing application of the Downstream platform. The goal of this application is to generate a trend analysis of air quality over a specific area. The application can generate these trend analyses for five gas types. The user can then derive the air quality from those trend analyses. The gas types supported are nitrogen dioxide (NO2), sulfur dioxide (SO2), ozone (O3), formaldehyde (HCHO) and aerosols.

The application makes use of the dataset generated by OMI, Ozone Monitoring Instrument [31]. This instrument is aboard the eos aura satellite which was launched July 15th 2004 and is operated by NASA [32]. This satellite makes 14 to 15 orbits per day. This way OMI can cover the whole earth each day, as each orbit only a section of the earth can be covered.
The data generated by OMI is publicly available and comes in the HDF5 format [29]. Many data products are available. The level 1 dataset contains the calibrated geo located measurements derived from the raw data. In this product the individual gasses are not split up yet. The level 2 dataset contains the data for each individual track over the earth for each gas. It is derived from the level 1 dataset. The level 2 gridded (L2G) dataset combines the tracks of each gas in full daily orbits. The measurements are put onto a grid of 0.25 by 0.25 degrees latitude and longitude [33]. The air quality application makes use of the level 2 gridded data product. The datasets selected are OMTO3G for ozone, OMNO2G for nitrogen dioxide, OMSO2G for sulfur dioxide, OMHCHOG for formaldehyde and OMAERUVG for the aerosols.
The L2G dataset consists of a single HDF5 file per day per gas. The OMI datasets start from 1 October 2004 and are still generated to this day.

This application is a good match to base the evaluation on as it has an important change coming up. A new instrument called TROPOMI, TROPOspheric Monitoring Instrument, will be launched. This instrument is aboard the Sentinel-5 precursor satellite which will launch somewhere in 2017. This instrument can measure the same gasses as OMI did and more. The difference is that TROPOMI can provide a much larger resolution compared to OMI. This means that the area for each measurement is much smaller. This of course leads to much more data to process. As previously there was one measurement for an area, and now there are multiple measurements for that same area. It is very likely that the existing (original) application cannot cope with such an amount of data within a reasonable processing time. This makes the application ideal for big data processing, as it can scale out if the data amount, and thus the processing need increases.

The application itself is fairly simple. The application is written in Python and is split up into two parts, closely resembling the pre-processing with an on demand part structure we used for the architecture. The goal of the pre-processing step is to transform the HDF5 L2G files that OMI provides into a more simple format, and leave all the unnecessary data out. The first step of the pre-processing is to read the HDF5 L2G files. These files contain a lot of data besides the data we want to extract. For example the files contain data about the satellite for each measurement. If the data is processed using multiple algorithms, the files will contain the measurements for all those algorithms. Finally a lot of quality flags are present. While some quality flags are used, many of them can be ignored. Using some of these flags we can apply additional filters

to the data to mark some measurements as unreliable. A lot of the measurements are also filtered out in this step due to missing measurements. These missing measurements can be caused by environmental conditions causing the instrument to fail to measure, but mainly due to the structure of the HDF5 file. The HDF5 file contains data for each 0.25 by 0.25 degree cell for each orbit. As each orbit only covers a section of the earth most of this data grid will be empty.
Due to some of this filtering it may be possible that there are no measurements for a grid cell for a day. This may cause unwanted outliers in the final product, the trend analysis. Therefore the temporal resolution of this trend analysis is brought down. The third step in the pre-processing is to combine each measurement of each grid cell for a single week. That means that instead of potentially 7 measurements per week per grid cell, a single measurement per grid cell is produced. The week system used are ISO weeks. An ISO week is a week which is defined to start on Monday. The first ISO week of the year is defined to contain the 4th of January. Therefore each year contains 52 or 53 ISO weeks. It is likely that when no measurement can be made for one day to environmental conditions this condition will also be present for the whole week. It is therefore still possible that a 7 day reduction to a single week still has no data. To solve this the application does an interpolation over the grid cells that do have data to come up with a data point for a cell that has no data.
The final step is to store these weekly measurements into an intermediate storage. Note that this process is repeated 5 times, one time for each gas type.

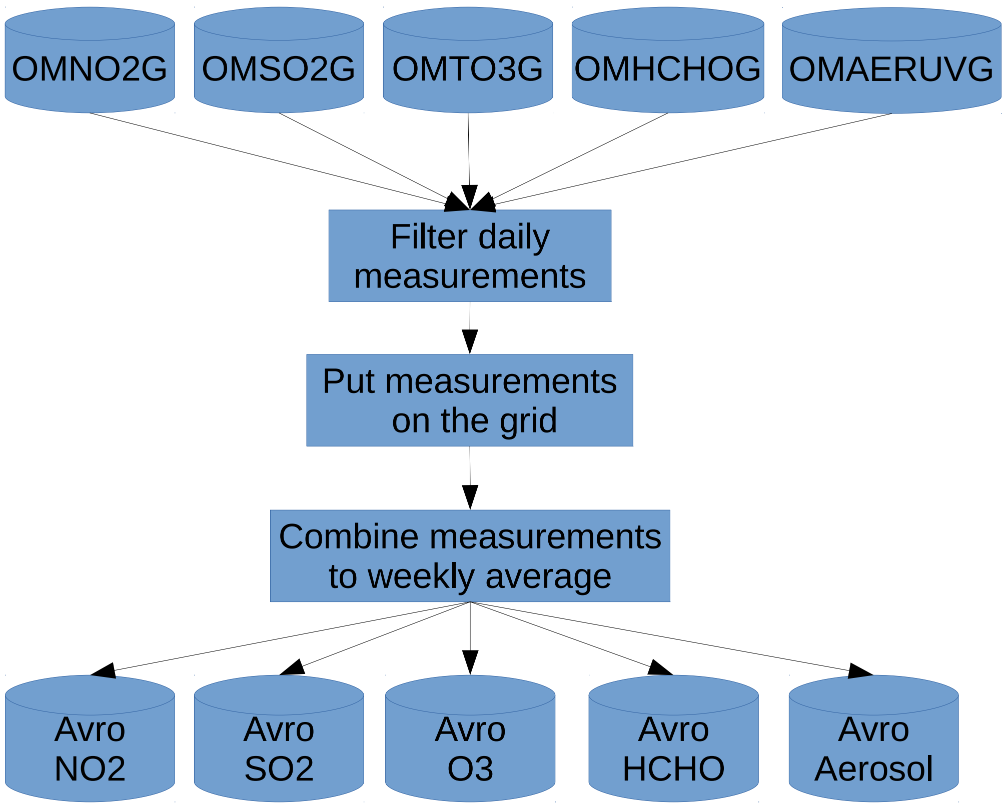

OMNO2G
OMSO2G
OMTO3G
OMHCHOG
OMAERUVG
Filter daily measurements
Put measurements on the grid
Combine measurements to weekly average
Avro NO2
Avro SO2
Avro O3
Avro HCHO
Avro Aerosol

Figure 4.1: Pre-processing step of the improved air quality application

Due to the pre-processing the on demand processing is very simple. The on demand processing has three parameters, the coordinates of the grid cell, the range of weeks and the gas type. The application will read in the appropriate intermediate weekly files according to the range of weeks and the gas type. The next step is to select the grid cell from this loaded data, and discard all the other cells. Finally the measurements are passed to the trend analysis algorithm. Note that in this final step we only have a number of measurements equal to the number of weeks we specified. The result of the trend analysis is plotted, and the plot is saved in a fixed location.

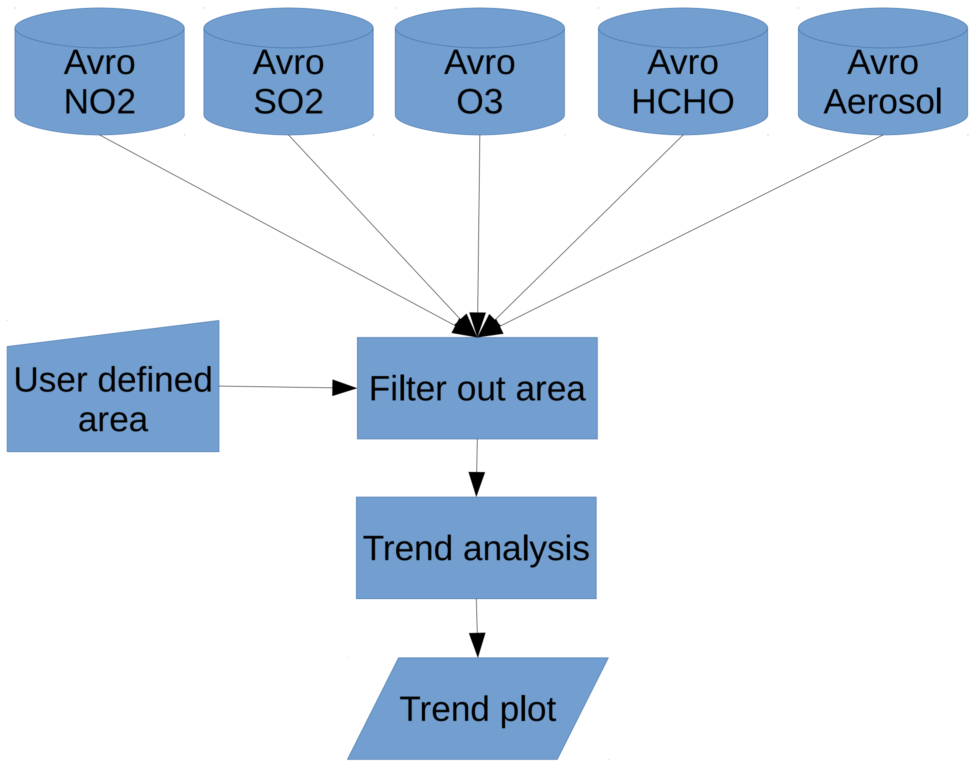


Figure 4.2: On demand step of the improved air quality application

4.2 Improvements over the original implementation

Since we are re-implementing the application for our big data processing platforms, we can adapt this application to simplify the implementation process and enhance the user experience a bit. All the enhancements are summarized in table 4.1.
The user experience is enhanced by combining the pre-processing calls into one. The original pre-processing processes one gas at a time. Since these gasses have no data dependency on each other, they can be processed at the same time. The new implementation of the pre-processing will read all the gas types at the same time, process them and output the results in gas separated directories. This way the end user only has to start the pre-processing one time instead of running it 5 times on the different gas types. Note that the on demand processing still does one gas at a time. If we would do the on demand processing for all 5 gasses at the same time, we might create a product that the end user

does not want, as he only requested one gas, this would therefore waste processing power.

The application is simplified for the implementation process in two ways. The first simplification is the resolution of the sulfur dioxide dataset. We mentioned that the level 2 gridded datasets contains data for each 0.25 by 0.25 degree cell. The OMSO2G (sulfur dioxide) dataset differs slightly in that it has an 0.125 by 0.125 grid cell size. That means it has four times more cells in the grid than the other datasets. The original implementation acknowledges this by having a separate processing line that does this processing on the four times larger grid. Since we now do the pre-processing over all gasses at the same time we want to generalize the process as much as possible. To do this we simplify the application by ignoring the higher resolution of the sulfur dioxide. We do this by reading the four measurements for the four 0.125 by 0.125 degree cells as if they were four measurements for one 0.25 by 0.25 degree cell. This doesn't cause any problems as the temporal reduction to a weekly measurement now reduces (up to) 7×4 measurements instead of (up to) 7.
Note that this means that the results of the application may differ for sulfur dioxide depending on which implementation is used. Is this a problem? No, this implementation is meant as a proof of concept, and not as an application that can go live at any moment.

The second simplification enhances the fact that the implementations are going to be different, and the results may thus vary. The original implementation does an interpolation over the grid cells to fill in a measurement for grid cells that have missing data. While this sounds good in theory, it doesn't really work in practice. In practice it is quite common that large sections of the world have no data points. With the interpolation solution grid cells in the middle of such a section get assigned a value that is based on actual measurements of grid cells that lie very far away from them. This is of course incorrect. To solve such problems, and simplify the application the interpolation process is removed from the new implementation. Note that this indeed gives different results versus the original implementation, as we now have grid cell that have no measurements, whereas in the original implementation all cells had a measurement.
This leaves us with the problem of grid cells having no measurement. We simply cannot solve this problem, as there is simply no data. We can make up some data as the interpolation basically did, but this will lead to incorrect data. The only thing we can do is try to avoid the missing points in the end trend analysis. We can do this by enlarging the area the trend analysis is done over. The on demand processing is adapted to instead of doing a trend analysis over on single grid cell over a period to a trend analysis over an area over a time period. We introduce a fourth parameter for the on demand processing, the area. In the new implementation we will define this area as a simple rectangle. The on demand processing will average the measurements of the grid cells in the given area to come up with a single trend analysis that represents that area. This will hopefully mask some of the grid cells in such an area having no measurement.

Table 4.1: Improvements of the new implementation for the air quality application

	Original	**New implementation**
Ability to process multiple gasses	Support 5 gasses. Processing one gas type per execution	Supports 5 gasses. Pre-processing of 5 gasses at the same time. On demand processing one gas type per execution.
Resolution of the data set	0.25 degree cells. 0.125 degree cells for sulfur dioxide	0.25 degree cells for all gasses.
Missing data points	Interpolate over the whole grid.	Remove the missing points. On demand processing works over an area instead of a single grid cell, to mask these missing points.

4.3 Introducing the map reduce model

Now that we have identified the steps of the application we can move towards the implementation. However before we can start implementing we first need to know the programming model used. As mentioned in section 3.2.2 the processing platforms use the programming model which is or based on map reduce. This model is inherently different from usual programming models as it focuses on scalability by exploiting data parallelism. This means that this model does processing on tiny amounts of data, but tries to do as much as possible of this processing on tiny amounts of data at the same time. In this way large datasets can be processed in a time that is very likely faster than by using a sequential processing platform.

So how does this work? The programming model has two main operations: map and reduce. The model assumes that we start off with a set of individual data elements. The map operation does exactly as described, it processes one single of this data elements to generate zero, one or more new data elements. The map operation is applied to all elements. This leads to a new set of individual data elements with a size that may be unequal to the original set. This process can be repeated multiple times.

At some point it is very likely that some data dependency is introduced in an application. That means that some of the individual data elements needs one or more of the other data elements to generate a new value. For example if you want to create a sum of a bunch of numbers, and each element is a number. You need a step where in order to create the sum element you need all the number elements, and not just one. The map function does not allow for such data dependencies as the input is only the single data element. The solution to this is the reduce function. Before the reduce function is called the data elements are grouped to form a set of groups from the set of data elements. The group a data element belongs to is defined by the contents of the data element, and is specified by the code. After grouping the reduce function is called. This function receives all the data elements of a single group and again produces zero, one or more new data elements. This process is repeated for each group in the set.

If we take all the map operations that generate the set of data elements B from set of elements A we see that none of those map operations have a dependency on each other. This means we can exploit the data parallelism. Each of those map operation can run at exact the same time. The whole process of generating set B from set A can take up as little time as the maximum time of one of those map operations. The same principle applies for the reduce operation, as the groups do not have dependencies on each other.

In figure 4.3 we can see the map reduce model with a single map operation when running on multiple servers. As the map operations have no dependencies on each other we can distribute the data elements in a random order over the servers, and schedule the map operation for that element on that server. The results of those operations are then stored on that particular server. The set of data elements that is produces by all the map operations is now distributed over all the servers. Here we see the potential of scalability. We can simply add another server, and when the processing is resubmitted some map operations that where earlier scheduled on of of the existing servers are now scheduled on the new server. Now the existing servers have less work to do, and the expected time to process becomes lower.

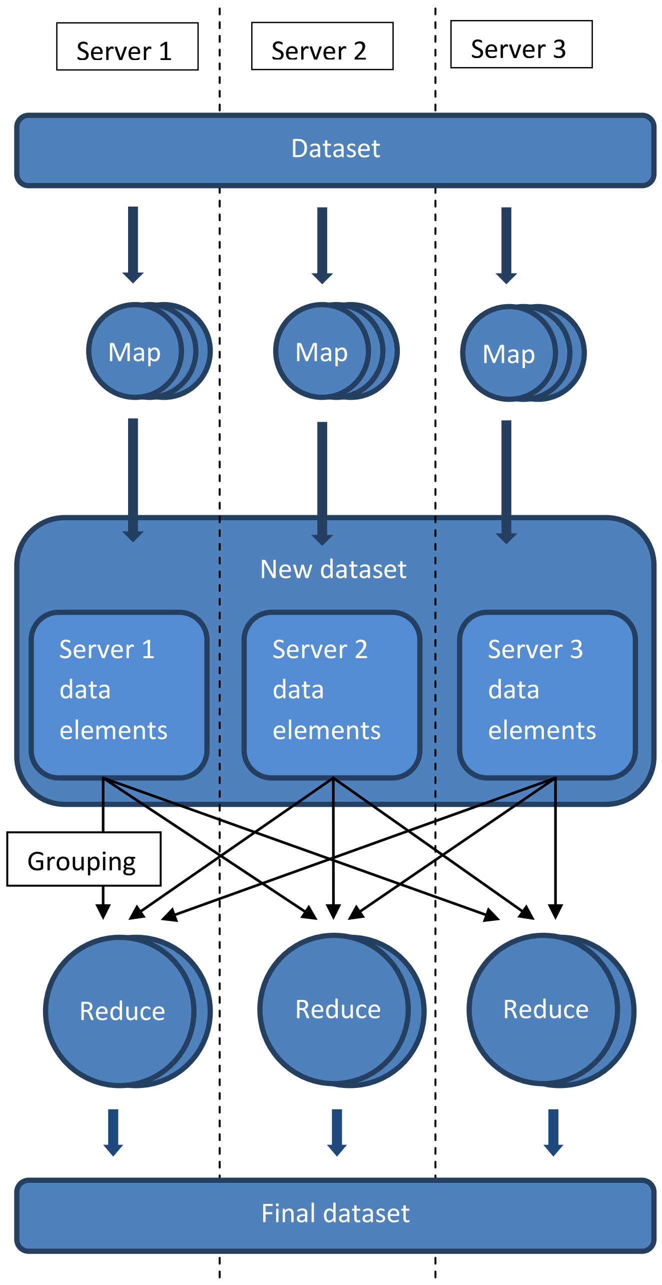


Figure 4.3: The map reduce model running on multiple servers

In the next step in the figure the elements are grouped. The reduce operations do not have dependencies on each other, and can thus run on any random servers. However we see the major performance slowdown here. Lets say an reduce operation is scheduled to be running on server 1. This operation requires all the elements in the group of that specific reduce operation. The data elements that belong to this group might however not be present on server 1. The map operations are potentially scheduled randomly, and so the resulting data can be present on random servers. To be able to start the reduce operation server 1 first needs to receive all data elements from the other server that belong to that group. This means that the reduce operation is much more performance intensive than the map operation.

4.4 Differences between the processing platforms

The original map reduce model that was introduced in 2004, which is implemented in Hadoop MapReduce, and later Hadoop MapReduce 2. Spark and Flink were introduced much later around 2014. It therefore makes sense that Spark and Flink try to fix any flaws that Hadoop MapReduce may have.

4.4.1 Hadoop MapReduce 2

Hadoop MapReduce 2 purely implements the map reduce model. It therefore has two operations, the map and reduce functions. What we did not discuss yet is that this original model also enforces the format of the data elements. The model requires that the data elements come in tuple format by using a key and a value pair. This means that the map operation takes a key-value pair and transforms that into zero, one or more key-value pairs. The generated key-value pairs all have to be of the same data type. The input key-value pair data type does not have to match the data type of the generated key-value pairs. The grouping process of the data elements for the reduce is done by grouping all key-value pairs together that have identical keys. The reduce function then receives the common key and all the value for that group. The reducer again produces zero, one or more key-value pairs.
Note that this model does not always fit the application. For example in our summing of numbers application we do not have a key-value pair, we just have numbers. In Hadoop MapReduce this is usually solved by using the single data value as key and make the value of type NullWritable. The NullWritable is a special type that indicates that there is no data. However we can agree that this circumvents limitations in the model.

Another limitation of this model is that the order of the two operations is fixed. First a single map operation is started, then the reduce operation and the application should be finished by then. Hadoop MapReduce does allow you to skip either the map or reduce phase, but you cannot issue more than 1 reduce operation, or do a map operation after a reduce.

4.4.2 Spark

The limitation on the programming model are solved in Apache Spark. The programming model, or API, of Spark is based on, but not limited by the map reduce model. The Spark API still supports the key-value pair map and reduce operation in some form. The reduce equivalent grouping is done explicitly in spark by issuing a group by key operation on a paired dataset. This results in a dataset consisting of pairs of keys and their corresponding set of values. On this dataset a map operation can be done to imitate the reduce operation. The paired dataset is only one of the options of data sets available in Spark. Spark supports data sets of a single type, so not only key-value pairs. Spark also supports data sets of tuples in the form of a single type data set where the type is a tuple. The tuples can be of any length, so it is possible to have complete records as data elements that make up a data set. Grouping for such data types is still possible as Spark also has a group function that relies on a user based function.
The second problem of the map reduce model was the fixed structure of the map and reduce operation. We already hinted that a reduce in Spark can be done by a grouping

followed by a map operation. This would make a map-group-map structure. However Spark is not limited to such a fixed structure. Spark allows for the chaining of any operations, as long as the data types match the operation. For example you cannot do a group by key on a set consisting of numbers, as they do not have a key type.
Since there is no required structure in the operators, Spark can also introduce a lot of new operators. For example Spark has operators for joining different data sets, returning the distinct elements of a data set and a simpler operator for filtering data sets.

4.4.3 Flink

The API of Apache Flink for batch processing is very similar to the API of Spark. The Flink API supports most of the operations of Spark and also uses the grouping operator. The small difference between the API's is the absence of an explicit paired data type for Flink. Flink only supports data sets with a single data type. Note that this data type may be a tuple, so a tuple of length 2 still implies a paired set. So why is this different from Spark then? The Spark API has a method groupByKey that only exist for paired data sets. This method assumes that the first value of the pair is the key, and can thus group on that value. Flink does not have such a method, but can group data elements together. The grouping is done by a user defined function, which is applied to every element and returns a value. Every element that has the same function value is grouped together. In Spark a group resulted in a paired data set of key and the set of values. In Flink there is no real notion of the key, so the result of the group is just a data set of the sets of grouped together values. Note that grouping of a key-value pair is still possible, but a bit harder. We can group a data set with data type tuple of length 2. The grouping function simply returns the first value of the tuple, which we said is the key. The key-value pairs are now grouped by our key. Next we can apply a map function to imitate the reduce function of map reduce. This function will receive a set of grouped tuples. If we need the key of that group we can just take the first value of one of the tuples, as they should all have the same first value, as that is our key.

4.4.4 Data storage in memory

Another interesting difference between the platform is how they deal with data that is larger than the available memory. For big data processing it is very common that the size of the data set is much larger than the available memory. Still sometimes the data has dependencies on other data elements, so that the complete data set is required in the system. The solution is to store the partially processed data on disk. Disks are usually much bigger than the available memory, and can thus store the data. Hadoop MapReduce writes in between the map and reduce operation the data back to disk. The reduce operation then reads back those partially processed data sections as needed. Of course this can harm the performance as disks are usually much slower that the memory. Spark and Flink try to store as much of the partially processed data in memory. This is also why they claim that their system is much faster. If the data exceeds the memory available, the remainder is spilled to disk.
Related to this is the way that the partially processed data is written to disk. The processing platforms need a way to get the data that is in memory to a format on disk and back. This serialization and deserialization in Spark is done by either the built in serializer of the programming language, or the Kryo serializer. Flink uses a custom

serializer. Note that these serializer do an automatic translation of the in memory objects to bytes, which can be written to disk. Hadoop MapReduce has a different solution. It has a Writable interface, which has methods for reading the object from bytes, and writing the object to bytes. All key and value data types are required to implement this interface. Then the system can simply call the write method on the object to generate bytes that represent that object, and write those to disk. In reverse the bytes from disk can be passed to the read method, which fills in the data for that data type. MapReduce has a lot of standard writable implementations for data types such as numbers, text, booleans and many others. However if the user defines it's own data types he has to implement this interface itself.

4.5 Data granularity

An important consideration when implementing an application on a big data platform is the granularity of the data. The term granularity refers here to the size of the data elements we pass to the processing algorithm. For example the air quality application we can pass each individual measurement to the map operation. We can also adapt the algorithm to take in all the measurements of a section of the world at a time. Note that still all the measurements will be passed to the processing algorithm, just the size of the batches will be increased. When a bigger granularity is chosen, and thus bigger chunks of data are passed to the processing at a time the amount of overhead that is caused by having many data elements is likely to decrease. On the other hand when dividing the data into larger chunks, the total amount of chunks decreases. If the granularity gets too big, there comes a point where there are not enough chunks to be distributed among the processing tasks, and thus some processing power remains unused. From a programming point of view the granularity requirement is application dependent. For example if the goal is to filter out a specific area, a low granularity may be easier. Using a low granularity you can filter out each data element at a time, instead of having to check each measurement in each chunk that is passed. Other applications may prefer a high granularity.
The air quality application that is implemented using the various processing platforms makes use of a very small granularity. We take the smallest data element we can find, which is the individual measurement per grid cell. This means that a single record of position, measurement and time is passed to a mapping task at a time.

4.6 Transforming the application into map reduce operations

It is clear that all three the processing platforms can in some shape of form perform the map and reduce operations. It would therefore make sense to try to transform the steps of the processing of the application into those operations.

4.6.1 Transforming the pre-processing

Let us start with the pre-processing steps. We have identified the steps taken by the application. When we apply the improvements we will make to the algorithm we get the following steps:

1. Read the HDF5 data files for all gasses
2. Filter out grid cell that have no measurement
3. Filter out low quality data
4. Generate grid coordinates
5. Combine all measurements in an ISO week per cell
6. Write the result to an Avro file for each week and gas

Note that step 4 was not mentioned before. All cells in the grid have coordinates of their location in the whole grid. We can read this coordinates from the HDF5 files. We also can read the exact latitude and longitude of the measurement from this file. Note that these coordinates do not always perfectly match the centers of the grid cells. This extra step ignores the grid coordinates from the input file and creates its own by deriving them from the latitude and longitude. Note that the latitude has a range of -90 to 90 degrees and the longitude -180 to 180 degrees. The method of doing so is specified by the following pseudo code:

```
$glat = floor((latitude + 90)/0.25)$
$glon = floor((longitude + 180)/0.25)$
```

This should give grid coordinates roughly equal to the original. So why is this used, and not just the original coordinates? In section 4.11 we will adapt the algorithm to use a different grid cell size. We do this in a very simple way by only having to adjust the coordinate calculations we specified by the pseudo code.

Now let us get back to the translation of those steps. Step 1 is the way to get the data into the processing platform. This is neither a map or reduce operation, but depends on the capabilities of the processing platform to read files. The process of reading the HDF5 files is explained in section 4.9.
Step 2 and 3 can be easily translated into map operations. The map function takes the measurement and quality data as data element that is produced by step 1. If a value is not passing the filter the map function outputs no new data element. If it does pass the filter the same data element it received as input is returned as output. Most of the quality flags are not used any more in the next steps, so they may be omitted from this new output data element.
Step 4 can also be represented as map operation. This map function requires the latitude and longitude from the measurement. Using this coordinates and the method of calculating the grid coordinates discussed before this map operation can output these grid coordinates.

In step 5 the first data dependency is introduced and should therefore be handled by a reduce function. In order to generate the weekly combined measurement for a cell we require all the measurements of that cell that have a time that lies in that ISO week. The obvious grouping would therefore be based on the combination of the cell coordinates and the ISO week the measurement was generated by OMI. However we should not forget that all the gasses are processed at the same time. We do not want to create a weekly measurement for all gasses combined per cell, but a weekly measurement per cell per gas. Therefore we need to add the gas type to the grouping key. The grouping is thus based on the combination of the cell coordinates, the week and the gas type.

In the original applications the reduction of multiple day measurements to a single weekly measurement was done by the same interpolation that handled the cells without measurement. Since we removed that part of the algorithm we have to come up with a new combination algorithm. The original interpolation was a weighted linear interpolation where the weight depended on the distance of the center of the cell to the latitude and longitude of the measurement. This concept is copied to the new algorithm. The new algorithm is basically a weighted average, where the weight is again dependent on the distance of the center of the cell to the latitude and longitude of the measurement. This averaging is described in pseudo code in listing 4.1.

Listing 4.1: Combining measurements into a weekly average with a weighted average

```
 1  function reduce(glat, glon, week, group) {
 2          centerLat = glat * 0.25 + 0.125 - 90;
 3          centerLon = glon * 0.25 + 0.125 - 180;
 4
 5          sumWeight = 0
 6          sum = 0
 7          for(latitude, longitude, measurement in group) {
 8                  weight = 0.3 - (abs(centerLat - latitude) +
 9                          abs(centerLon - longitude))
10                  sum += weight * measurement
11                  sumWeight += weight
12          }
13          return sum / sumWeight
14  }
```

Note that the constant 0.3 on line 8 was picked semi random. The maximum deviation of the center of the grid cell in degrees is 0.125 for the latitude and 0.125 for the longitude. This makes a total maximum deviation of 0.25 degrees. Since we do not want to give these edge measurements a weight of 0 the constant 0.3 was picked. The more this constant comes closer to 0.25 the harder deviations from the center of the cell will be punished with a lower weight. The actual value of this constant does not matter though, as again, this application is a proof of concept, and the results do not match with the original implementation anyway.

In this processing step another type of weight is also produced. Besides the measurement per week per cell per gas a weight is also given for each cell for each week for each gas. This weight is independent from the weighted average weight and represent the trustworthiness of this weekly measurement. This trustworthiness is derived from the number of measurements in that week and the uncertainty of the measurements given for each measurement in the HDF5 files. Note that only nitrogen dioxide and formaldehyde have those uncertainties. For all other gasses only the number of measurements is used. The algorithm for this weight calculation is copied from the original implementation. Let us say that in eq. 4.1 N is the number of measurements and u is the set of uncertainties.

$$w = \frac{N}{\sum_{i=1}^{N} u_i \times u_i} \tag{4.1}$$

For all the other gasses that do not have uncertainties eq. 4.2 is valid.

$$w = N \tag{4.2}$$

In the final step, step 6 we write the weekly measurement and weight for each cell in the whole grid for each gas to the Avro based intermediate storage. For clarity each Avro file will contain all the measurements in the grid. A file will be generated for each week and each gas. The process of writing the Avro files is described in section 4.10. What we are still missing for this process to happen is the transformation of the data set of weekly measurements per cell to a data set that contains weekly grids of measurements. It is possible that grid cells that belong to the same week and same gas type were processed on a different server. Therefore we first need to issue a reduce step to group the measurements together that will be stored in a single Avro file. We can do this by grouping on the week and gas type. The reduce function will output these groups, so that they can be stored as the Avro files.

To determine what data we need in every step we backtrack the flow of the data. Note that as the week is represented as an ISO week and the data spans multiple years, the data type used for a week is a combination of the week number with range 1 to 53 and the year. Any further mention of the week data type is this combination.
Step 6 requires the week, the gas type, the grid coordinates, the (weekly) measurement and the weight.
Step 5 requires in addition to the data of step 6 the latitude, longitude and the measurement uncertainties. This step generates the weight needed in the next step. The uncertainties are no longer needed in the next step.
The process of putting the measurements back on the grid of step 4 only needs the latitude and longitude. This step generates the grid coordinates. Since the next step also requires the measurement, uncertainties, week and gas type these are also a requirement for this step.
The second filter of step 3 only needs the quality flags. It also requires all the requirements of the next step.
The first filter of step 2 only requires the measurement to detect if the actual value is missing. It also requires the quality flags, uncertainties, latitude and longitude, week and gas type for the next steps.
This leads to a translation to the following steps:

1. Read data
   Output: (latitude, longitude, week, gas, measurement, quality flags, uncertainty)
2. Filter (map)
   Output: (latitude, longitude, week, gas, measurement, quality flags, uncertainty)
3. Filter (map)
   Output: (latitude, longitude, week, gas, measurement, uncertainty)
4. Grid (map)
   Output: (latitude, longitude, week, gas, measurement, uncertainty, glat, glon)
5. Combine (reduce, group on (week, gas, glat, glon))
   Output: (week, gas, glat, glon, measurement, weight)
6. Output as Avro (reduce, group on (week, gas))

4.6.2 Transforming the on demand processing

We can apply the same process to do the transformation to map reduce steps for the on demand processing. The algorithm has the following steps:

1. Read the weekly measurements and weights for the range of weeks and the specific gas type specified by the algorithm parameters.
2. Filter out any grid cells that do not lie within the area specified by the algorithm parameters.
3. Average all measurements and weights in the area to a single value per week.
4. Do a trend analysis over the averaged values and output the plot.

In step 1 the weekly Avro files are read. The process of doing so is described in section 4.10. Where in the pre-processing we passed the complete grid to the Avro writing process, we read the individual grid cells back in the on demand process.
In step 2 we again have a filter. Since the Avro reading process outputs a single grid cell as data element we can use a map function that uses the grid coordinates of this grid cell to determine if the cell is within the specified area.
The first data dependency is introduced in step 3. In this step we combine all the measurements and weights that where not filtered away in the previous step by week. We can do this by using a reduce operation. First the data elements are grouped on the week. Now we can do a simple average of both the measurements and the weight in the reduce function. Note that the weight is averaged, and has nothing to do with an weighted average. The weight is only used in the trend analysis itself.

The final step is the trend analysis itself in step 4. To implement this we have to make an important decision. Can we, and would it be useful, to implement the trend analysis using map and reduce functions? The algorithm for creating a trend analysis requires all the data points to be available. The algorithm itself is also sequential. Theoretically it is possible by grouping the whole dataset together and doing the trend analysis in a reduce function. However implementing such an algorithm is not an easy job. We already have an implementation which does the trend analysis (using a library) and creates the end result plot, which is the original implementation. The trend analysis is therefore implemented as an output for the big data system that simply pipes all the weekly averaged values to an external Python script, which contains the analysis and plot code from the original. So is this piping process not very expensive in terms of performance? No, the result of the averaging step is just one measurement and one weight for each week. Let us make an estimation of the data. The OMI data starts from 2004. Let us ignore that some years have 53 ISO weeks, and say each year has 52 week. For 2016 we then have a maximum amount of $52 \times 12 = 624$ weeks. We can thus say that the amount of week, measurement and weight pairs passed to the external python process will stay well under a thousand. With modern hardware this is not a problem.

The backtracking process to determine the data types is very simple for the on demand processing. The trend analysis output of step 4 requires the measurement, weight and week. The averaging of step 3 has the same requirements. The filtering step requires in addition the grid coordinates of the cell. Note that the indication of the week still refers to the combination of week number and year. This leads to a translation to the following steps:

1. Read the weekly measurements.
   Output: (glat, glon, week, measurement, weight)
2. Filter area (map).
   Output: (week, measurement, weight)
3. Average (reduce, group on week).

Output: (week, measurement, weight)

4. Output to trend analysis.

4.7 Translation of map reduce operators to the processing API's

Now we have the description of the application in map and reduce operations we translate them into the correct API calls of the processing platforms. This will create the major part of the implementations. The other part is the interaction with the HDF5 and Avro files, which are explained in section 4.9 and 4.10.

The map operator can be translated into API calls called map and flatmap in Spark and Flink. The difference between these calls is that flatmap resembles the map operator we described which return zero, one or more new elements. The map API call always returns one new element. Some of the map operators can also be translated into the filter API call, which acts as the filter we described. When the data element passed to the filter function may pass the filter it is returned as new data element. When the data element does not pass the filter no new data element is generated.
The map operator can be implemented in MapReduce 2 by implementing the Mapper interface. This interface acts exactly as the description of the map operator.

The reduce operator can be, as described earlier, implemented in Spark and Flink by issuing the group API call and then a map API call. In MapReduce 2 we use the Reducer interface. This function this interface describes acts again as the description of the reduce operator. Note that for MapReduce 2 we need to define the key-value pairs. The reducer groups on the key values of the input pairs. We can therefore define the key types as the combination of the values we wanted to group on. The value of the key-value input pair of the reducers can be the remaining fields.
Note that we also need to define the key-value input and output pairs for the mappers. The actual data used for key and value does not really matter in the mappers, as the key is not used for anything special like with the grouping of the reducer. It makes sense to try to keep the key and value constant over the operations, as that might save some data copying and is easier to implement.

4.8 Problems

4.8.1 Python

When implementing the application some problems were encountered. Let us start with the biggest problem: Python. One of the important aspects we defined in section 2.2 was that is should be easy to implement new applications. One of the measures to do so would be the use of the Python programming language. Each processing platform we picked supports Python in some shape or form. However when implementing it became clear that this support was a bit lacking. For example MapReduce 2 is based on Java. It supports Python by allowing the Mapper and Reducer to be implemented in among others Python. The runtime itself is still Java, this means that you still need a Java implementation to tell the system you are using a Python script as mapper or reducer. The data itself is also situated in the Java memory space. This means that a translation is needed to a format that Python can understand, and therefore copy all the data to the Python script. This process is of course not good for the performance. Also this does not make it easier to create new applications, it only makes it harder versus a pure Java implementation.

Spark and Flink have an API that is pure Python. However it looks like that the Java versions are better optimized. When testing the implementation of reading in the HDF5 files in Spark it looked like Java was much faster. The testing set up consisted of a single HDF5 file containing the OMNO2G data set. All measurements where read in, and as result a simple count of the amount of measurements was made. The Python implementation took over a minute, where the Java implementation took only 10 seconds. Due to these problems with Python it was decided to do the implementation in Java for each platform.

4.8.2 The Hadoop MapReduce fixed structure

We already mentioned the second problem before. Hadoop MapReduce 2 only supports applications that follow the strict format of a map operation followed by a single reduce. If we look back at our map and reduce operations for the pre-processing step we see that we have a structure of three map operations followed by two reduce operations. The three map operations can be dealt with. Hadoop MapReduce 2 has a method of calling multiple map operations from a single mapper by chaining the calls to the map operations. The reduce part however is problematic. The only way to create an application with multiple reducers in MapReduce is to chain two applications together. The first application does the reading of the data, the filtering and the process of putting back the measurements on the grid. Finally this application does the temporal reduction to a single week. The results, that is the individual pairs of measurement, grid coordinates, gas type, week and weight for each week and grid cell, are stored on disk. The second application reads these pairs from disk, and does a reduce to group those pairs by week and gas type. Then finally the grouped results are sent to the Avro writing process. Note that the second application also needs to follow the map reduce structure, and therefore needs a mapper. To do so a map operation is used that outputs all the input data elements it gets, also known as the identity mapper.

4.8.3 Polymorphism and Hadoop MapReduce writable

In section 4.4 we already discussed how MapReduce requires user defined objects to implement the Writable interface. For the air data, being the measurements, week and quality flags combination such a object was used. The measurements and week is used for all gasses, where the data types for the quality flags depends on the gas type. As mentioned before we use Java to implement the application, which is object oriented. This allows us to define a parent class which defines the measurement and week data fields. A class for all the gasses exists which extends this class and adds new data fields, as required for the gas type. Due to the polymorphism of Java we can address the gas specific instances as if they were the generic parent class. Let us say the parent class is called AirData and two of the gas specific child classes are called NO2AirData

and SO2AirData. Let us look at a map operation. The map operation requires that the input and output types are defined. Only one type can be defined. This means that if we would define the input type as NO2AirData we would not be able to pass SO2AirData to it. In, for example, the first filter of the pre-processing we want to filter all gas types. Due to this polymorphism we can define the input type as AirData, while the actual instances may be of type NO2AirData or SO2AirData.
The challenge lies in the fact that Hadoop MapReduce requires the classes to implement the writable interface. This means that the object should be able to write itself to a byte stream, and read its contents from a byte stream. If we define an output type as AirData the platform expects that the AirData type can read and write itself. The actual instance type is however not AirData by the gas specific types. The writing part is not the problem here. The specific classes can overwrite the writing method of the AirData class to also write their gas specific data fields. This can also be done for the reading method, but this will never be called. The platform expects that a AirData type is written and later read back. It does not know which specific instance type is written. When reading back the data it therefore creates an AirData instance and calls the read method. As the instance types are lost, the generic AirData type can not read the gas specific fields. As the gas specific fields are written as bytes, the reading process will interpret those bytes wrongly as if they where of the AirData fields, and thus read back incorrect data. Note that this is not a problem in Flink and Spark, as those serialization mechanism do store the actual class that was written, and can thus correctly create instances of the correct child classes. The solution is not very elegant. A wrapper class is used that contains the AirData class as field. This wrapper implements Writable. When the write is called a byte is added to indicate which gas type the wrapper contains. When reading back the platform creates an instance of the wrapper and calls the read method. This method reads the gas type byte and creates the correct instance type for that gas itself. Finally it calls the read method on that new instance to read the actual data.

4.8.4 Hashcode inconsistencies

The final interesting problem is that of hashcode inconsistencies. For the reduce operator a grouping of values is done. As mentioned, to do such a grouping often a key value is used. Any values having the same key are grouped together. In Java the equals function exists, which is implemented by the user. Implementing such a function is very simple, as it only has to compare the fields of the object. As you can imagine calling the equals function for each object is not very fast though. To check on a whole set to see if any of the elements are equal to each other you require $\frac{1}{2} \times N \times N - 1$ operations on a set of length N. As data sets in big data often overpass billion of elements, this would take way to long. Processing platforms solve this by using a hash code. A hash code is the result of a function, the hash function, that can convert an object to a number. The conversion is constant, so each object where equals would return true results in the same hash code. Note that the result is not unique, so different objects may map to the same hash code. If the platform first applies the hash function, we can do a quick rough grouping on that number. Finally we have to call the equals function for each object within the group, as different objects may result in the same hash code. These groups are much smaller, so calling equals for all objects does not nearly take as long as on the whole data set.
The interesting part here is that it is very important to implement this function, even

when the default implementation would usually suffice. The default implementation generates a hash code that is derived from the memory address of the object. For objects such as the AirData this does not suffice, as the many instances of these objects do not have the same memory address. The object that does define the type of the gas in the grouping on the week and gas type this should suffice. The gas type object can be constant, and reused in every week and gas type combination. As it is reused the memory address also stays the same, and the default implementation should suffice. What we forgot here is that we can, and will, use multiple machines. While the gas type is constant, each machine has its own instance and thus a different memory address. Mistakes like these lead to hash codes being different on different machines for the same object. This caused objects not being grouped together, and therefore giving inconsistent and wrong results.

4.9 Reading HDF5 files

We have determined how the data is processed, but not how the data is read into the system. The HDF5 file format is an hierarchical format containing groups, data sets and meta data [29]. The whole file itself is one group. Each group can hold both other groups and data sets. The data sets can be accessed via the names of the groups, just as if its was a UNIX directory tree. In the data sets we can find meta data describing the properties of the data data and a (multi dimensional) array containing the actual data. As we need multiple values like the latitude, longitude and measurement, we need to read multiple data sets and combine them into individual pairs that can be used in the processing chain. We already defined all the values we needed: latitude, longitude, week, gas, measurement, quality flags and uncertainty. The actual names of the groups and data sets for those values differ for each gas type. However they can be extracted from the original implementation.

Since the actual format of the file is quite complicated an external library is used to do the reading of these HDF5 files. The HDF group provides a Java library that can be used. The problem with this library is that it is basically a wrapper around the C HDF5 library. This means that a native library needs to be accessible by the Java virtual machine on each machine running a worker. This limits the portability quite a bit. The other problem is that this native C library uses the standard C library to read the raw bytes of the HDF5 files. We want to use the Hadoop Distributed File System as a storage solution for the HDF5 files. The standard C library does not have support for this file system.
A better library is the NetCDF library [34]. The NetCDF4 file format is based on HDF5, so any library that can read NetCDF4 can read HDF5. The library does not require any external native libraries. While the library does have its own byte reading model, it does provide reading HDF5 (or NetCDF4) from a byte array. With some adjustments this was transformed into an implementation that forwards the raw byte read calls to the Hadoop Distributed File System API.

The implementation of reading the data elements is done using the Hadoop MapReduce InputFormat. The implementation of the InputFormat specifies which RecordReader should be used. A new RecordReader is created for each input file. The RecordReader has three main functions: nextKeyValue, getCurrentKey and getCurrentValue. The

nextKeyValue function should read the next key and value pair from the format this RecordReader implements, in our case the OMI HDF5 files. The getCurrentKey and value functions then return the key and value of the pair that was just read.
The OMI HDF5 reader parses the whole file when created. The correct groups and data sets are loaded. Each nextKeyValue call the next value from the various data sets we selected is retrieved and combined to form the key and value. Note that as the groups and data sets differ for each gas, each gas has its own RecordReader implementation. The correct RecordReader is selected by the name of the HDF5 file, which contains the name of the data set as a whole. This can be used to trace back to the gas type. For example the data set name OMNO2G leads to the nitrogen dioxide gas type. The other names are mentioned in section 4.1.
The nice thing about the InputFormat implementation is that it is also supported by Flink and Spark. This way we only need one implementation for all systems. This also means that the performance influence of reading in the data should be stable for the three systems.

4.10 Interaction with Avro

The final part of the application is the serialization and deserialization of the intermediate data between the pre-process and the on demand processing. For this process we are using the Avro file format. From the transformation of the application to the map and reduce operators we can see what data will be stored in the intermediate format. The final step of the pre-processing is to take the combination of week, gas, glat, glon, measurement and weight and group on the combination week and gas. We therefore get groups of glat, glon, measurement and weight which we can store in a file per week and gas combination. This means that the on demand processing can select the weeks and gas by loading the respective files.

Apache Avro has built in support for Hadoop MapReduce [35]. This support is however lacking and is not used. For the on demand to be able to only load the files it needs, we need the output process being able to write multiple files. These files should also be named so that it contains the week and gas type. The built in Avro support follows the default output mechanism. Each file created is named part-r-xxxxx where xxxxx is the number of the task of the reducer that created the data. This also means that each reducer only creates one file. In this file the set of key-value pairs outputted by the reducer will be written. In our case the output of the reducer is a key with type (week, gas) and the value is the set of grouped (glat, glon, measurement, weight) pairs. The behaviour of writing all key-value pairs to a single file is thus definitely not what we require.
The solution is to write a custom OutputFormat that writes a file per (week, gas) key it receives. The file path and name of this file is equal to *[output folder]/[gas type]/[year]/[gas-type]-[week number]-[year].avro*. The writing process itself is done via the Avro Java API.

To be able to read and write data in Avro, a description of the structure of the data is required. In Avro this is called a schema. For the intermediate data the schema in listing 4.2 is used.

Listing 4.2: Avro schema for the intermediate data

```
{
"namespace": "nl.airbusds.airquality.common.data",
 "type": "record",
 "name": "ProcessedAirData",
 "fields": [
     {"name": "gridx", "type": "int"},
     {"name": "gridy", "type": "int"},
     {"name": "measurement", "type": "float"},
     {"name": "weight", "type": "float"}
 ]
}
```

From this schema a Java class is generated using the name as class name and the namespace as package. This class contains the data fields described, and the functions to build the object, which are used by the Avro reading process. The problem with this generated class is that it is required to implement Writable for MapReduce. As this class is generated, adding code to implement Writable would be bad, as it would be overwritten every time the class is regenerated. The solution to this is again a wrapper around the class that implements Writable.
Both Spark and Flink also support this custom OutputFormat, therefore the implementation can be shared for all the implementations.

The reading back in of the data in the on demand processing is again done using a shared custom InputFormat and RecordReader. This implementation reads for each file the week from the file name to create a key value. Note that the gas type is ignored from the file name as the on demand processing only can do one gas at a time. The selection of the input intermediate files determines which gas is used. The implementation next reads all the values of (glat, glon, measurement, weight) pairs from the file using the Avro API as the generated ProcessedAirData class. The output of the implementation is pairs of week and ProcessedAirData. Note that the week can be reused for each key-value pair for that file, as the week is constant for the whole file.

4.11 Scaling up to TROPOMI data

The current air quality application makes use of the satellite data generated by OMI, Ozone Monitoring Instrument. It is expected that this data set will be replaced by the one generated by the successor of OMI: TROPOMI. TROPOMI has a much bigger resolution compared to OMI, this means much more data to process. In order to see if the architecture can handle data sets with much higher resolutions the behavior of the application with respect to OMI versus TROPOMI data should be tested. The problem with this method is that the TROPOMI data is not yet available. The solution to this problem is duplicating the OMI input data. The resolution of TROPOMI is about 4 times larger in both the latitude and longitude directions compared to OMI. To emulate this the OMI data is duplicated about 16 times. The HDF5 files are read 16 times to simulate reading larger data files. For each cycle an additional data field containing the number of the cycle is added to the data elements. This cycle number is later used to determine which of the 16 sub cells of the OMI sized cell this data element will represent. This leads to the values of each OMI sized cell being copied 16 times to

create the TROPOMI sized cells. This gives an indication of the expected processing time of the real TROPOMI data.
Note that for simplicity the scaling up to TROPOMI is only implemented for the pre-processing and only for Spark.

Chapter 5

Results

5.1 Testing environment

5.1.1 Hardware

To be able to test we first need to figure out the hardware that will be used. Since no cluster is available that can be dedicated to testing, a cloud solution is used for testing the implementations. The Google cloud is is picked, as Airbus Defence and Space has already used this solution in previous projects.
Since the Google cloud is used, it makes sense to map the testing parameters on the available instance types. The standard instance types are available with 1, 2, 4, 8, 16 and 32 (virtual) CPU cores. In terms of scalability therefore two factors can be tested: The scalability in the number of cores, and in the number of nodes. For each implementation the following scalability tests are done: The effect of scaling by using more cores will be tested by running the test on a single instance. Each next test the instance type is changed to increase the amount of cores. This will be done up to 16 cores.
The scaling out will be tested by starting with a test on a single 4 core instance. Each test the amount of instances is increased by one, up to 16 instances. Note that both scaling methods will also increase the amount of available memory. The Google cloud instances come with 3.75 GB of memory per core. Increasing the core count, therefore increases the memory with 3.75 GB per added core. The 4 core instances contain 15 GB of memory. Scaling the amount of instances thus increases the total amount of memory with 15 GB per added node.

Besides the amount of cores, and therefore the amount of memory, a storage structure is also required. Each instance used for testing has it's own root disk. This disk is of the standard hard drive type and has a size of 200 GB. This disk is used for the operating system and the storage of the HDFS blocks. For the processing platforms this disk is also used to store the logs and temporary files, such as the files generated by spilling the data to disk when the memory is insufficient. Next to that disk a 500 GB standard hard disk exists. This disk contains a section of the data set generated by OMI. This disk is shared among all the instances. The Google cloud only allows the disk to be shared, if it cannot be written to the same time from multiple instances. Therefore this disk is attached as read only on all the instances. This doesn't matter as the input OMI data is only read, and never written by the workers.

Besides the worker nodes we have one other instance we need to create which is the head node. The task of the head node is to run the resource manager for YARN and the name node for HDFS. These tasks are not very heavy in terms of performance, as long as it is more or less constant for all tests. The head node is therefore ran on an single core instance type.
Note that in the description of the storage one component is missing. None of the disks contain the processing platforms themselves, their configuration and the implementations. The software of the processing platforms should be accessible by all nodes. However the configuration should be flexible in the way that it can change. In the implementation phase, and testing the implementation, it is possible that you would want to quickly change a configuration option. Therefore a read only solution does not work. The solution is to store the processing software, the configurations and the implementations on the root disk of the head node. The other worker nodes attach those directories via the notwork via the NFS protocol. As the files shared can be cached, this does not impact the performance in any significant way.

5.1.2 Software

All the instances, being the worker nodes and the head node, run on Linux using Ubuntu 16.04.1 LTS as the operating system. A small adaptation to the operating system was required to get a correct working system. Ubuntu, just as many Linux distributions comes with the kill command default installed. The kill command can terminate a process given its process id [36]. At the time of testing Ubuntu shipped with a faulty version of the kill command. Hadoop MapReduce 2 uses this kill command to terminate all the map reduce processes of a job after it finishing that job. It does so by issuing the kill command with process id -pid. This special syntax for the process id means kill all processes in the same group as that pid. The bug being that if the pid would start with a 1 the kill command interprets that as -1 and not as the full pid. For example kill -s 9 -1234 would be interpreted as kill -s 9 -1. The -1 option indicates to the kill command that all processes with a PID larger than 1 are signaled, and thus terminated. So if the process id of a map or reduce process happens to start with a 1, all processes on the machine are killed. Of course a normal user can not kill essential processes, but still the YARN and HDFS services are killed, which is unwanted. Debugging this problem is also difficult, as any ssh sessions and debugging programs are also killed. This problem was solved by replacing the kill command with a patched version.

All the software makes use of Java. It is therefore essential that that Java JRE and JDK are installed. The air quality indicator makes use of language features only available in the Java SE version 8. For example lambda expressions, which allows for easier declaration of the mapping and reducing functions, are used. To be able to compile and run the air quality indicator application therefore Java 8 or higher is required. Hadoop, Spark and Flink by themselves only require Java 7 or higher. The testing machines run OpenJDK-8-jdk update 111.

The whole Hadoop platform, including HDFS, YARN and MapReduce 2 comes in one package. The version installed on the testing machines is 2.7.3. Besides the configuration changes required to run MapReduce on YARN two other properties were changed. The property *yarn.scheduler.capacity.resource-calculator* is used to determine which resource

scheduler should be used by YARN. The default resource scheduler ignores the amount of cores available, and schedules purely on the available memory. This means that if only 1 GB of RAM is available an allocation of a worker requiring 2 GB will be denied by the resource scheduler. Since we want to increase the amount of cores in the tests, we also want to limit the allocations by the amount of cores available. Therefore this was changed to *org.apache.hadoop.yarn.util.resource.DominantResourceCalculator*. This scheduler limits allocations based on both memory and the amount of cores.
The second property changed is *yarn.nodemanager.vmem-pmem-ratio*. By default YARN monitors both the physical memory and the virtual memory a process uses. Since the worker process will have many input files open at the same time, the virtual memory limit is reached very fast. When this limit is reached YARN kills the worker. To prevent this this limit is increased from 2.1 times the physical memory to 5.1 times the physical memory.

For spark the version installed is 2.0.1. The only configuration change required is the serializer as mentioned before in section 4.4. The property *spark.serializer* is set to *org.apache.spark.serializer.KryoSerializer*.

For Flink version 1.1.2 is installed on the testing machines. No additional configuration is required.

While testing some additional problems related to the memory allocations showed up. The default configuration for Spark and MapReduce 2 is a maximum total memory allocation of 1 GB for both the job master and workers. For Flink this limit is set even lower, the default maximum total memory allocation is 256 MB for the application master and 512 MB for the worker. This is simply not enough, and when processing enough data the workers would frequently crash with an OutOfMemoryError or the YARN monitoring killing the worker for allocating more memory than allowed.
Both Flink and Spark spill their results to disk if they do not fit in memory, so shouldn't they be fine with less memory? Indeed the processing part is fine with less memory. The problem lies in the reading of the data. The reading of the HDF5 files requires a lot of memory to parse the files and keeps lots of the data in memory.
Since the process of reading in the data is very crucial for the application, this problem can not be circumvented. The only solution is to increase the amount of memory that is available to the processing. To make a fair comparison the memory of all the systems was increased to the same amount. All the application workers get a limit of 2 GB of memory per core. The memory for the application masters is not that important. The only job of these masters is to control the workers. Since this function is not impacted by the implementation or the actual application running, the default amount of memory should suffice. This memory size also fits with the instance types. The single core instance has 3.75 GB of memory. With a single worker running of 2 GB and an application master using 1 GB, 0.75 GB is left for the operating system. Any larger limits on the allocation for the worker will probably stop the system from working correctly, as not enough may be left for the operating system and the application master.

5.2 Additional tests

It is expected that the processing time slightly depends on non deterministic factors like for example process and network scheduling. Since the Google cloud is used, some influence from other users on the same physical node is expected. It is therefore likely that the processing time will differ slightly each run. To test the influence of these factors the pre-processing is ran a couple of times spread out over multiple days. This gives us an estimation of the expected error percentage of the measurements. Note that running all the tests multiple times can not be done. The tests run in terms of hours, this is simply too long, so repeating them all will take way too long.

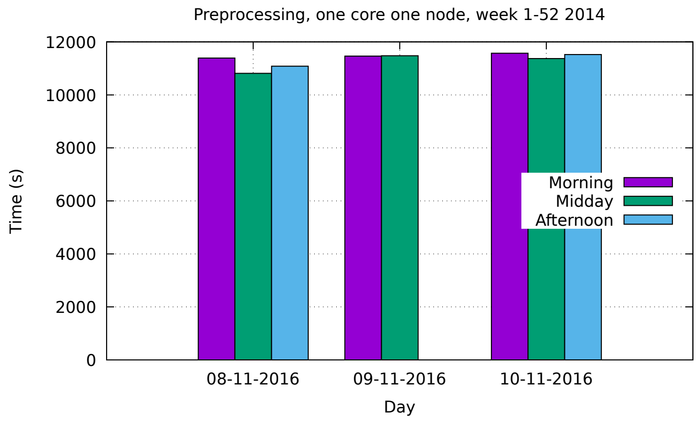
The input data is stored on a single drive that is accessible from all worker nodes. While this is easier for administration, as all data is in one central place, this may harm performance. The goal is to have many worker instances do parallel computing. This also means that the workers will be reading in the data in parallel. This leads to many instances asking for data from the same disk, which leads to poor performance. A better solution is to store the input data closer to the worker nodes. By using the same HDFS installation used for the intermediate results, we can distribute the data over the worker nodes. We can test the difference between the HDFS input data, and single disk solution by running the pre-processing applications for both situations.

As mentioned in section 4.1, the application that will be used for testing, the air quality application, is based on an existing application. An important aspect for Airbus Defence and Space Netherlands is the comparison of the new architecture to the existing solution, being the existing Downstream platform. The goal for Airbus Defence and Space Netherlands is to improve the existing solution. To see if the new architecture is indeed an improvement, the air quality applications on both platforms can be compared. Comparing these two based on the qualitative user needs may be useless, as the user needs where not around when the first implementation was made, and thus will this implementation very likely not comply to these needs. A comparison will be made on the quantitative needs. Those needs are performance and scalability. It is however important to note that the implementations of the existing and new implementation differ. Some errors in the existing solution were fixed in the new implementation. This also means that the algorithm executed by both implementations differs, and can thus perform differently. We have to keep this in mind when comparing the quantitative needs.

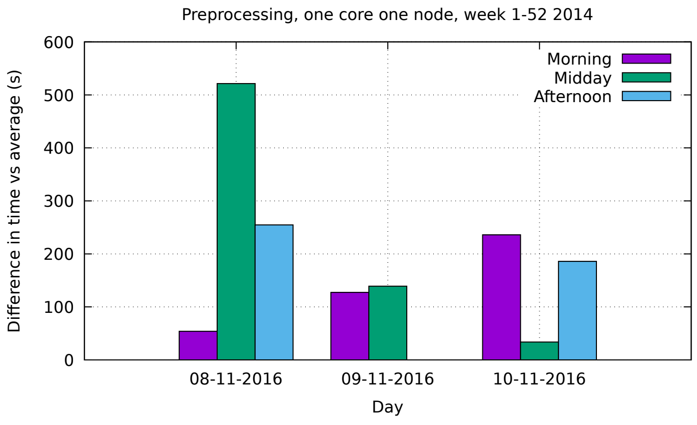
5.3 Stability of measurements

The first tests measures the stability of the measurements. This is tested by running the pre-processing step for the OMI data of all gasses. The range of the input data is limited from ISO week 1 of 2014 to week 52 of 2014. The resulting intermediate files are stored in HDFS. As mentioned in chapter 4 no front end is implemented. The processing is therefore started by executing the appropriate commands for the processing platforms from the command line. The processing time is measured from the start of the execution of the command to the command returning a value of 0. The return value 0 indicates the command ran successfully, and thus the processing ran without encountering any problems. This set up and data range is used in most of the tests.

For the stability test this pre-processing step is executed three times per day for three days for the Spark implementation. After each day the worker nodes are shut down, and new fresh instances are created for those workers each morning. The Google cloud service is popular, therefore it is likely that Google has many servers that host the instances. It is also expected that many instances are created and destroyed by other users. It is therefore likely that when the stability testing instance is destroyed, and later recreated, it will be running on a different physical server than before. Using this we can see if the performance changes when running on different servers. As the exact inner workings of the Google cloud are hidden, we cannot be certain that this happens though.
The three parts of the day, morning midday and afternoon, where chosen to see if the performance differs depending on the part of the day. Web based services often use the most processing power in peaks. This is usually when the targeted audience is the most active. If the physical server is shared with such an application the performance may vary for the three parts of the day. The tests are executed using a single node using a single core. This gives us the largest change that the physical machine is shared, as more smaller (less core) virtual instances fit on a physical machine. The results can be seen in figure 5.1a and table 5.1.



(a) Measurements on different days and times



(b) Difference in time versus the average

Figure 5.1: Stability of measurements

As we can see the processing most of the time takes over 11000 seconds, which is over three hours. One measurement sticks out, which is midday on 08-11-2016. This measurement is much lower than all the other measurements. Note that 09-11-2016 has no afternoon measurement. This was caused by accidentally killing the process. As there was no more time for another measurement that day, this field was left blank.

Table 5.1: Measurements on different days and times

Day	Morning	Midday	Afternoon
08-11-2016	11390.24	10814.91	11081.56
09-11-2016	11463.59	11475.46	
10-11-2016	11572.31	11369.95	11522.28

In figure 5.1b we can see the difference in time of the measurement versus the average, calculated as $abs(measurement - average)$. Again we see the spike of midday on 08-11-2016. This measurement differs 521.38 seconds from the average of 11336.29 seconds. This does sound a lot, as it differs almost 10 minutes. However compared to the measurement this is only 4.82 percent of the measurement itself. From these measurements we can conclude that the performance of the pre-processing on the Google cloud is relatively stable. As mentioned before, due to time constraints, measurements for the tests cannot be repeated. Running the tests repeated times to average the measurements will simply take too much time. Therefore in the results of the remaining tests we have to keep in mind that the measurements have about a 5 percent error margin.

5.4 Application footprint

Using the implementation we can retrieve some interesting numbers while testing. For example we can see how much data elements are in the data set at each processing step using these numbers and the estimated size in memory of those elements we can make an estimation of the application footprint in memory. In table 5.3 we can see these numbers for the pre-processing as extracted from Spark for the year 2014 for all gasses.

Table 5.2: Size of the data elements in memory

Field	**Type**	**Size (bytes)**
lat	double	8
long	double	8
measurement	float	4
week	byte	1
year	short	2
gas specific		6
Input element		29
glat	int	4
glon	int	4
gastype	byte	1
Filtered element		38
glat	int	4
glon	int	4
measurement	float	4
weight	float	4
Reduced element		16

Table 5.3: Number of data elements present after each step, week 1-52 2014

Number of input elements	34,434,201,600
Number of present elements	1,045,546,181
Number of filtered elements	571,142,343
Number of reduced elements	129,620,738

Using the estimated size of the elements in memory of table 5.2 we can see that the input elements for 2014 take up an estimated $34434201600 \times 29 = 998591846400$ bytes. This

roughly equals 930 GB for just one year. It is very likely that this amount of memory does not fit in memory, so spilling to disk is necessary.
Using these numbers we can also see hoe effective the filtering is. The number of present elements represent the amount of elements after the filtering of step 2 of section 4.6.1. We can see that over 90 percent of the input elements are actually not usable data. From those elements only a little over 50 percent of the elements survive the filter based on the quality flags of step 3. This leads to only 1.66 percent of the original input data being used.
After the filtering and putting back the measurements on the grid we have a memory footprint of $571142343 \times 38 = 21703409034$ bytes. This is about 20 GB of data for a single year.

The temporal reduction reduces the amount of elements to 129,620,738. These elements only contain the grid coordinates, the weekly measurement and the weight. The intermediate data, which is the output of the pre-processing contains these elements. Using table 5.2 we can see that these elements will take up an estimated $129620738 \times 16 = 2073931808$ bytes. This roughly equals 1.9 GB of data. This does not sound like a lot of data. Is the intermediate storage necessary? Could we not keep this data in memory? We have to keep in mind that this is just for the year 2014, if we say the whole data set is about 10 years we get about 19 GB of data. This still does sound reasonable. However we have to keep in mind that this is just one application, if we had multiple applications that kept 19 GB of data in memory, this would fill up the memory very fast. It is also likely that some applications have a much bigger footprint. For example if we would look at the scaling up from OMI to TROPOMI data we would expect an increase of 16 times the data, and thus 16 times the intermediate data. Suddenly we have an application with a memory requirement of over 300 GB of memory for 10 years of intermediate data. These amount of data are very likely to not fit in memory.
A second reason why the intermediate results can not be kept in memory is due to the way the processing platforms execute jobs. A soon as a job is finished, the processes that where executing the job are killed. Any results in memory are thus lost, as soon as the processing stops. Even if the processes where kept alive, the platforms separates jobs, so an on demand job can not access the memory of a pre-processing job. This is of course logical in a security sense of way, but stops applications sharing results in memory.

5.5 Comparison to original implementation

The next test we will run is comparing the performance of the old sequential implementation to the new big data based implementation. To be able to compare the implementations, a slight different parameter set compared to the Stability test is used. Both implementations select the data files by year and ISO week. The difference is that the original implementation requires the data files, whereas the new implementation silently ignores missing daily files. On the testing server the data for years 2014 and 2015 is present. Week 1 of 2014 however contains days that lie in 2013 due to the definition of an ISO week. When running the original implementation on the input data range of the stability test, it will notice that some data files for ISO week 1 of 2014 are missing and therefore return an error. To solve this the range of the input data is changed for both implementations from week 2 of 2014 to week 1 of 2015. As the testing environment

contains all the input files for these weeks, the original implementation can run. Since the input range is changed for both implementations, the amount of data read and processed is exactly the same for both.
Another requirement for running the original implementation is the gas selection. The original implementation can only process one gas at a time, where the new implementation can process all 5 gasses at the same time. To make a fair comparison both implementations only process NO2. As the original implementation is purely sequential, the test is done using a single core on a single node. In figure 5.2 we can see the results of running both implementations.

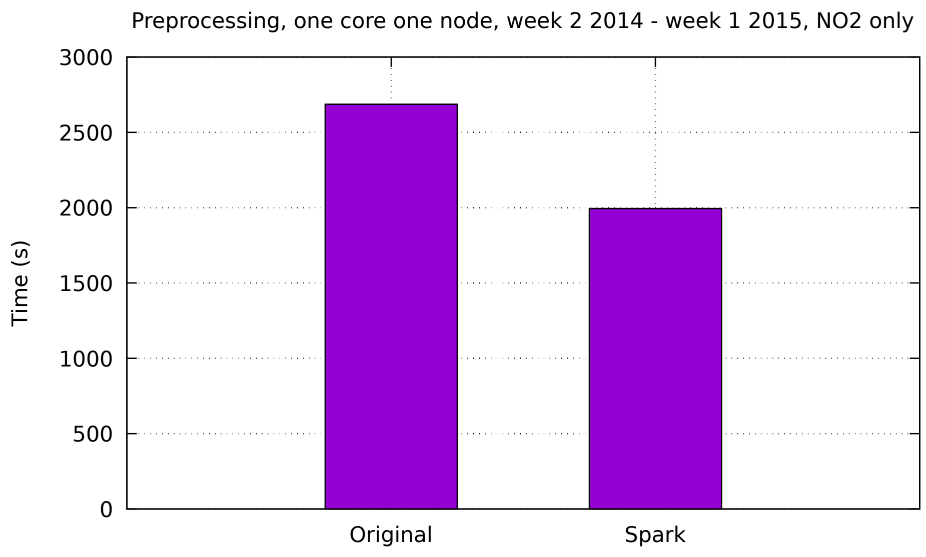


Figure 5.2: Comparison of performance versus the original

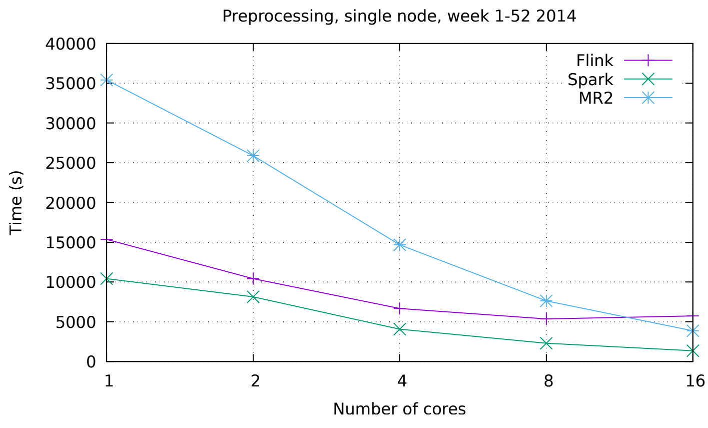
From this data it is very clear that the new implementation is faster than the original with a speedup of over 1.3 without even scaling up or out. This speedup is caused by two things. The first one being the new implementation is much more optimized than the old one. The old implementation was much more focused on getting the application working, rather than optimizing. The second part is the improvements made in the new implementation as mentioned in section 4.2. The old implementation used an interpolation over the whole grid, which is removed in the new implementation.
It is expected that new implementation can achieve an even greater speedup by applying the scaling it was designed for.

5.6 Pre-processing

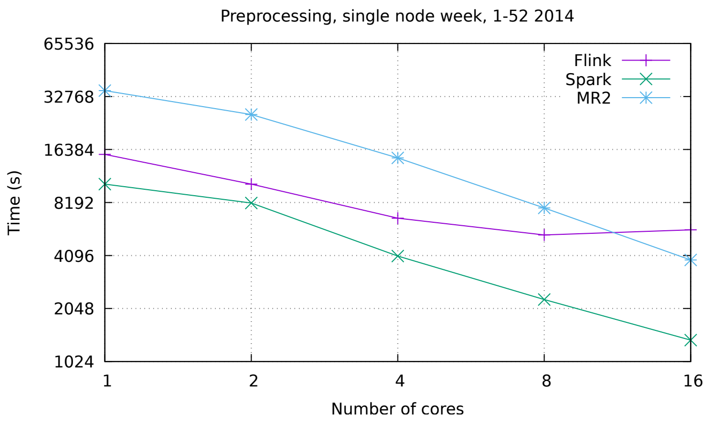
5.6.1 Scaling the number of cores

In figure 5.3a and 5.3b we can see the measurements of the pre-processing for the three processing platforms. The set up for this test is almost identical to the stability test. The processing time is measured for all gasses for the ISO week 1 of 2014 to week 52 of 2014. The raw data can be seen in table 5.4.
From first sight we can see that the implementation running on Spark is the fastest for

all number of cores. The MapReduce implementation seems to be by far the slowest. The single core test shows the MapReduce implementation being almost 3.5 times slower as the Spark implementation.
An interesting observation can be made for the measurements of the Flink implementation. The processing time for the 16 core test is higher than the 8 core test. Even if we would apply the 5 percent error rate in the best possible outcome for Flink, being the 8 core being 5 percent too low and the 16 core test 5 percent too high, the times would be around the same.



(a) Measurements for different core counts



(b) Measurements for different core counts on a log_2 scale

Figure 5.3

Table 5.4: Measurements for different core counts in seconds

#cores	Flink	Spark	MR2
1	15364.82	10419.43	35411.05
2	10422.82	8141.46	25886.46
4	6674.34	4065.48	14676.54
8	5361.92	2303.52	7626.45
16	5732.85	1354.09	3863.56

In figure 5.4 and table 5.5 we can see the speedup that occurred when using more cores. For Spark and MapReduce we see a line that looks straight. This indicates that the speedup is linear within error margins. Note that the speedup is by no means perfectly linear, meaning that doubling the amount of cores does not decrease the processing time by two. The tests show only a maximum speedup of 9.2 when using 16 cores.

Table 5.5: Speedup for different core counts

#cores	Flink	Spark	MR2
1	1	1	1
2	1.47	1.28	1.37
4	2.30	2.56	2.41
8	2.87	4.52	4.64
16	2.68	7.69	9.17

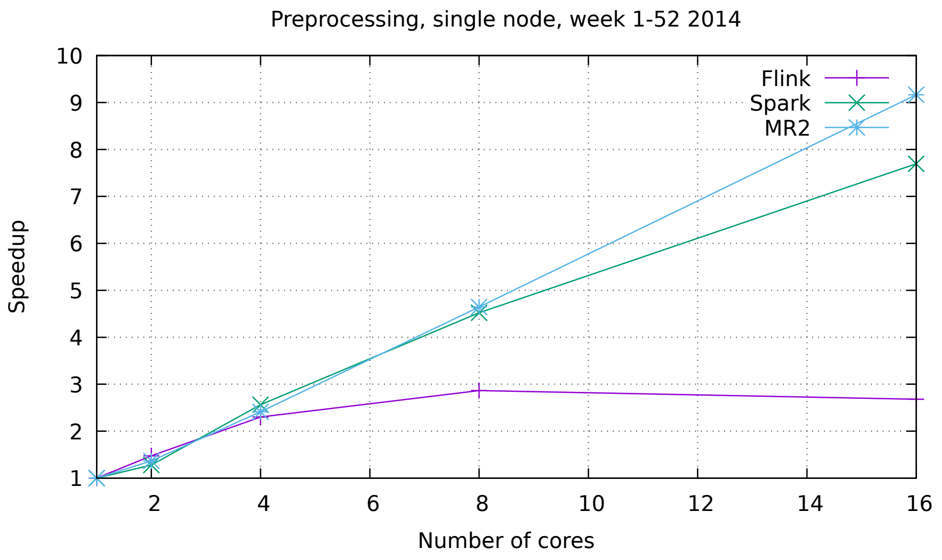
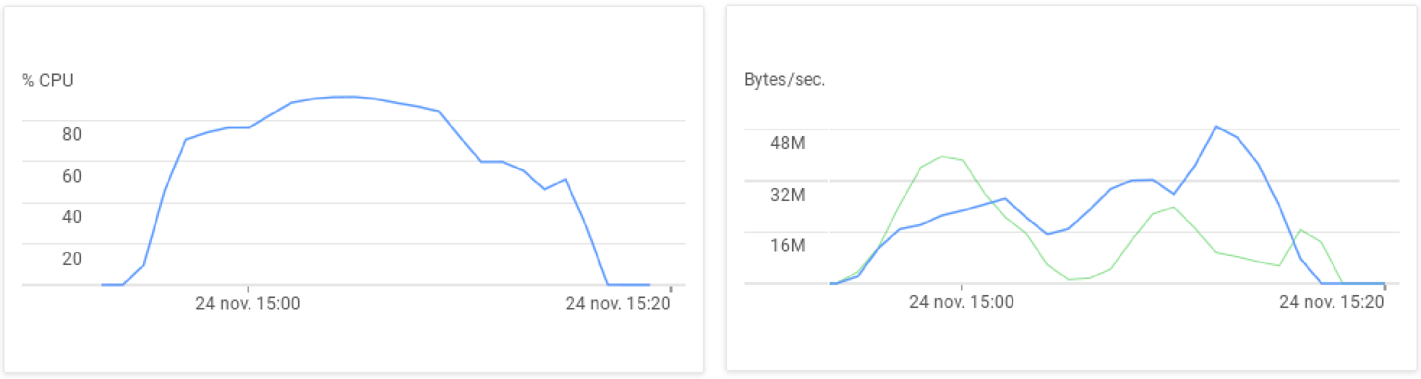


Figure 5.4: Speedup for different core counts

This figure also shows the strange behaviour of Flink. While the speedup seems to be linear for 1 to 4 cores, it flattens out when more cores are added. This can be caused by two things. The first possibility is either the processing platform or the implementation of the application does not scale as expected. The second possibility is that the processing is limited by a resource that does not scale with the amount of cores.

To further investigate the issue with Flink, we can look at the machine usage as provided by the Google cloud.



(a) CPU usage

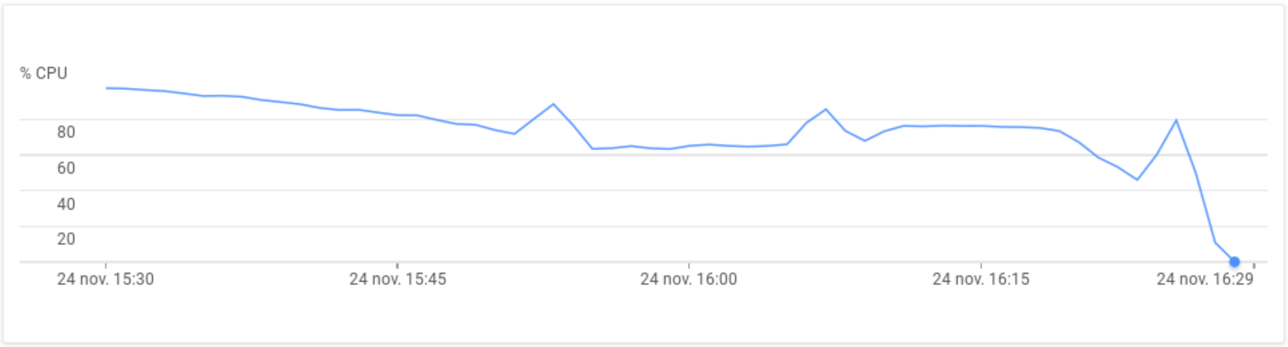
(b) Disk usage, blue is read speed, green is write speed

Figure 5.5: Worker machine usage for pre-processing in Spark using 16 cores

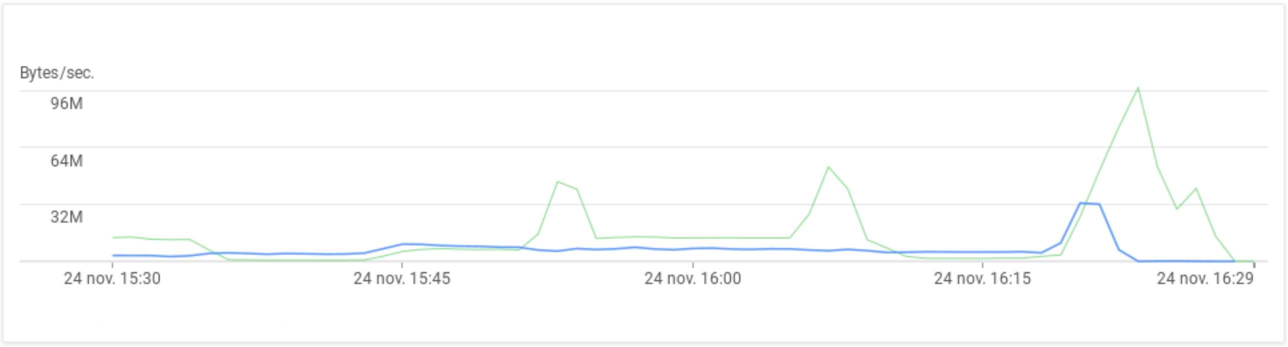
Machine usage for Spark

In figure 5.5a and 5.5b we can see the CPU and disk throughput of the Spark implementation running the same test as the core scaling using 16 cores. For the CPU usage a average of all cores is shown meaning that a 100 percent usage means all 16 cores are working 100 percent.

From the I/O graph we can see that Spark splits the processing into three parts. The first part is the reading of the HDF5 files, filtering and putting the measurements on the grid. The second part is the temporal reduction. The final part is the grouping for and writing of the intermediate files. After each phase the results are spilled to disk, as we can see in the spikes of the write speed.



(a) CPU usage



(b) Disk usage, blue is read speed, green is write speed

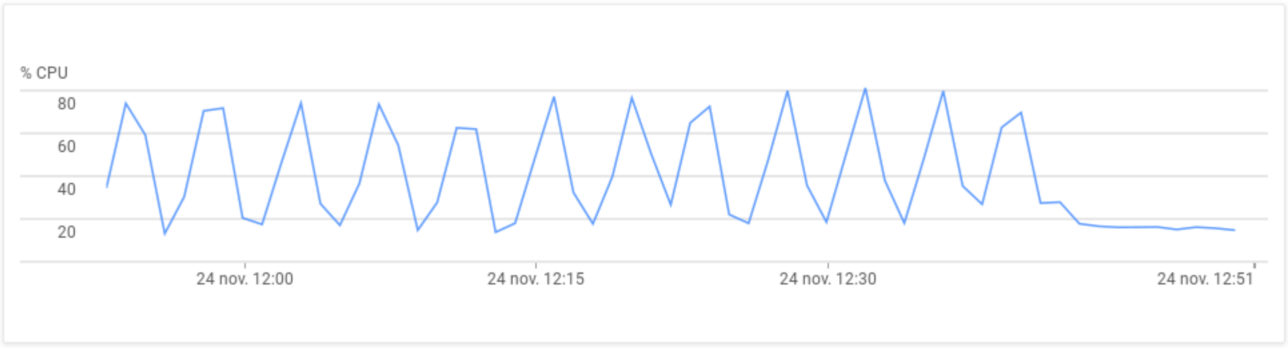
Figure 5.6: Worker machine usage for pre-processing in MapReduce 2 using 16 cores

Machine usage for MapReduce 2

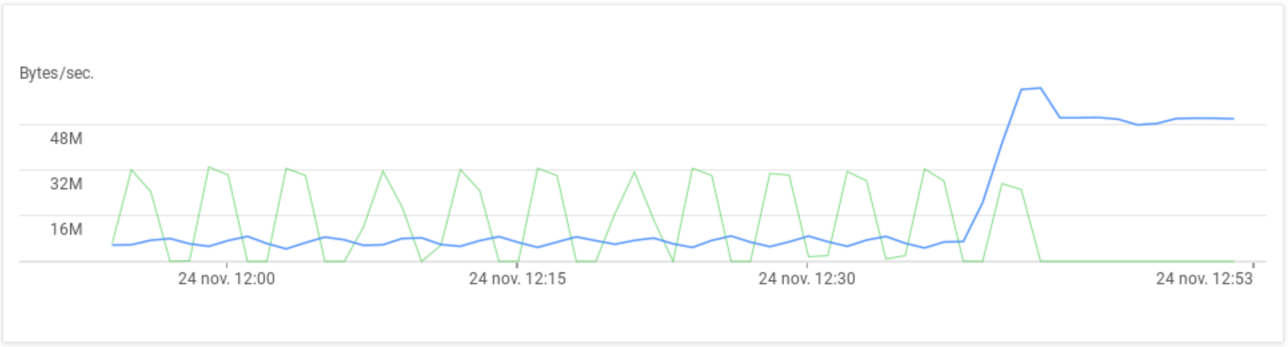
In figure 5.6a and 5.6b we can see the CPU and disk throughput of the MapReduce 2 implementation. We can clearly see a big difference versus the Spark implementation. Note that as the processing time is over an hour the start is cut off.

To explain the difference we first have to look at how tasks are executed on both processing platforms. Spark requests resources from YARN, then starts an executor process with those resources. Tasks are pushed to this executor process from the application master. The executor does the work and asks for more work. In MapReduce 2 each task is a separate process. This means that the application manager requests resources for each task process. As the resources are limited only a couple tasks get granted the resources they need. All the other tasks are queued, so when a tasks finishes YARN allocates those resources for a new (queued) task, which can then start doing its work. Two tasks exists for MapReduce 2 being a map task and a reduce task. The MapReduce 2 system starts issuing reduce tasks after it figures that enough map operations have finished. The reduce tasks get priority in the queue, probably to get the final results out as fast as possible.

The problem with this structure is figuring out when enough map operations have executed to start the reducers. In the CPU graph we see that over time more reducers are allocated. These reducers apparently did not have enough input yet, as they remain fairly idle, and thus the CPU usage goes down. The spikes in CPU usage (and I/O) are the moments where the reducer starts processing. The final spike in CPU and disk usage is the secondary application which groups and writes the intermediate output.



(a) CPU usage



(b) Disk usage, blue is read speed, green is write speed

Figure 5.7: Worker machine usage for pre-processing in Flink using 16 cores

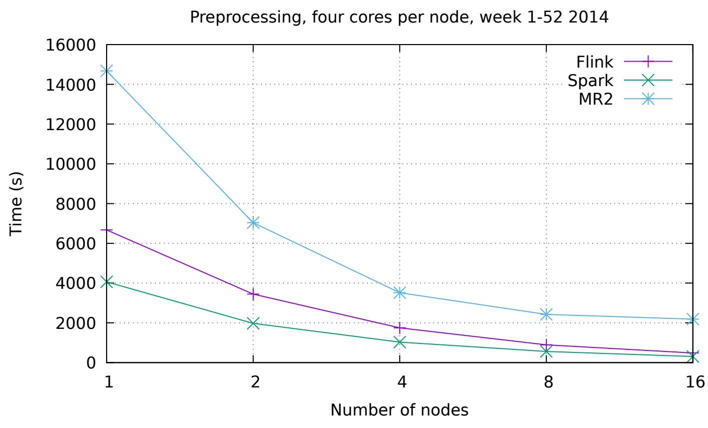
Machine usage for Flink

In figure 5.7a and 5.7b we can see a section of the CPU and disk throughput of the Flink implementation. We can see strange pattern of the CPU usage and disk throughput spiking. It seems like the CPU usage spikes roughly at the same time the disk write throughput spikes. The disk read throughput is slightly increased when the CPU is mostly idle.
This pattern appears at the first stages of the processing, so the platform is most likely reading the input data and doing the filtering and spilling the data to disk. This indicates that the system can function at high usage when it has data for a while, next it has to wait for data from the disk, and the usage drops to minimum. This means that very likely the input disk reading throughput is limiting Flink. As the disk throughput does not scale with the amount of cores, this limiting factor becomes larger and larger, which can be seen in the speedup graph of figure 5.4.

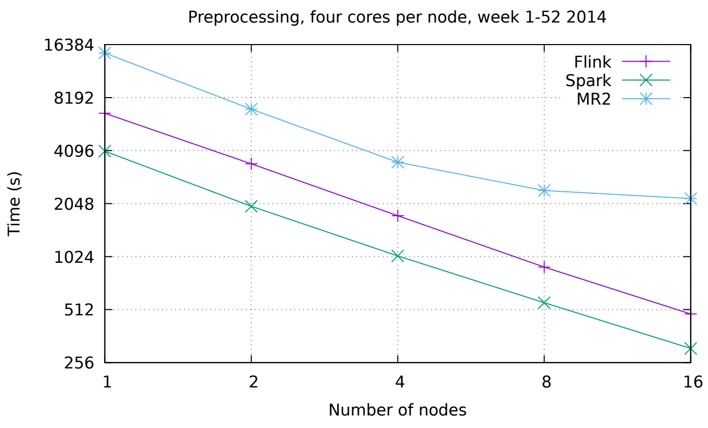
5.6.2 Scaling the number of nodes

The next step is testing the scalability for scaling out. The testing parameters are almost identical to the test of scalability in the amount of cores. The amount of nodes is doubled up to 16. The worker nodes each have 4 cores. This was chosen because most modern CPU's have 4 cores. The results can be seen in figure 5.8 and table 5.6.

We can see that the scaling for Spark and Flink is much better for scaling the amount of nodes than the amount of cores. If we compare the result of Spark in the graph with log_2 time scale in figure 5.3b with figure 5.8b we see a big difference. The cores graph has a clear bend, the scaling in nodes gives a straight line.
The line for MapReduce 2 is not straight at all, which indicates that it does not scale as well as Spark.



(a) Measurements for different node counts



(b) Measurements for different node counts on a log_2 scale

Figure 5.8

Table 5.6: Measurements for different node counts in seconds

#nodes	Flink	Spark	MR2
1	6674.34	4065.48	14676.54
2	3446.79	1976.32	7042.25
4	1746.11	1032.75	3518.02
8	893.05	559.92	2428.21
16	483.25	308.26	2186.52

This is confirmed in the speedup in figure 5.9 and table 5.7. Where Spark and Flink seem to scale very well MapReduce 2 seems to flatten out in the speedup graph. This can be explained by an observation made while testing. It was mentioned before that each map task is an individual process in MapReduce 2. With 16 nodes, each having 4 cores, we have a potential 64 tasks running at the same time. In MapReduce 2 each input file results in one map task. Processing the reading, filtering and putting the measurements onto the grid takes about 10 to 15 seconds for a single file. With 64 potential tasks running at the same time, on average the system has to allocate resources for 256 map processes per minute. From observation of the resources allocated the system struggled to do so. A lot of the time cores were left not allocated, and thus idle. Of course this leads to severely diminished performance which can be seen in the testing results.

Besides the MapReduce 2 speedup we can also see that the speedup for Spark and Flink is not perfectly linear, as the line is not perfectly straight. It seems like the speedup per added node decreases slightly with the amount of nodes. This can be explained by the communication overhead. More nodes means more communication required, which leads to more overhead, and thus slightly decreased performance.

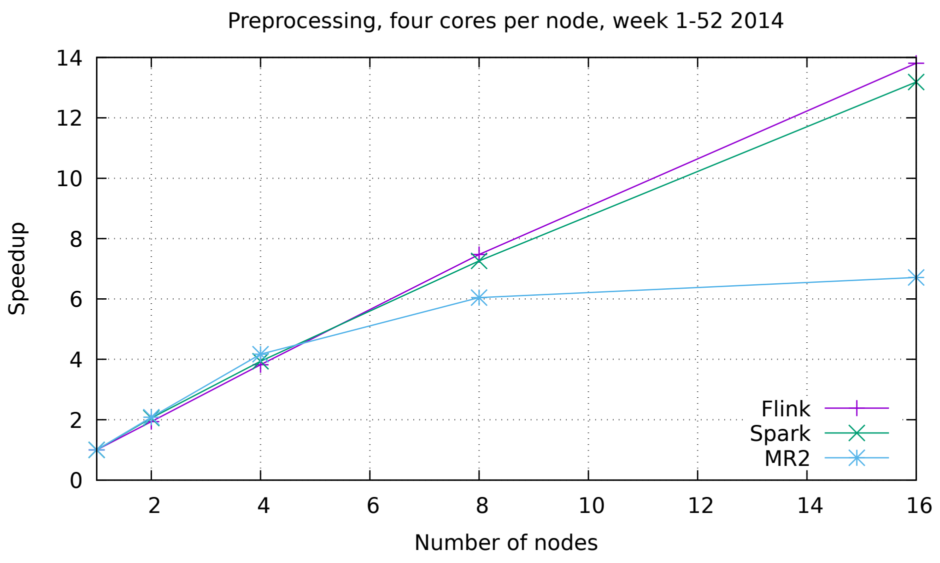


Figure 5.9: Speedup for different node counts

Table 5.7: Speedup for different node counts

#nodes	Flink	Spark	MR2
1	1	1	1
2	1.94	2.06	2.08
4	3.82	3.94	4.17
8	7.47	7.26	6.04
16	13.81	13.19	6.71

When comparing the speedup of the scaling of the cores versus the nodes we also see much higher speedups when scaling the amount of nodes. This goes against the prediction of section 3.3.2. We indeed see that the speedup goes down due to the network overhead, which the scaling in cores does not have. The scaling in cores however only gives a speedup of about 9 versus the single core solution. The scaling in nodes gives a maximum speedup of almost 14 times versus the single node (4 core) solution.
The explanation to this strange phenomena can be explained by the way the input data is stored. As mentioned before the input data is stored and read from a single disk. This disk is attached to all the worker instances as read only in the Google cloud. In the Google cloud the throughput of a disk is dependent on the size of the disk [37]. The default disk has a throughput of 0.12 MB/s per GB. This gives us a throughput of 60 MB/s. One would expect that when such a disk is shared among many instances, so

is the throughput. The tests in table 5.8 shows the opposite. For this test the Linux command *sudo hdparm -t /dev/sda1* was used. After each test all read caches were dropped using the commandos *sync* and *echo 3 > /proc/sys/vm/drop_caches*.

Table 5.8: Disk throughput for shared disk on different nodes

Read speed in MB/s	**Node 1**	**Node 2**
Not simultaneous	122.43	122.10
Simultaneous	122.49	125.39

The test shows that both nodes can read from the same shared disk at the same time at full speed. Therefore adding more cores does not increase the disk read speed, adding more nodes does increase the accumulated disk reading speed.

This also confirms the presumption that the Flink implementation was being limited by the disk reading speed. In the scaling nodes test the Flink implementation seems to scale slightly better than the Spark implementation. There is no more flattening of the speedup after adding more nodes, like the speedup of the scaling in the amount of cores did.

5.6.3 Using HDFS as input source

Another input source that we can use is HDFS, the Hadoop Distributed File System. HDFS is designed to store the data on the disks of the worker nodes. This can be used to have the data as close as possible to the machine where it will be processed, and thus lower data access time. This should result in lower processing times.

In figure 5.10 we can see that, surprisingly, using HDFS as input source does not result in a lower processing time. In contrary, the processing time is higher for all amounts of nodes when using HDFS versus the shared input disk. For the Google cloud this makes sense. HDFS tries to improve two things: access time by having the data more local to the processing nodes, and throughput by not having all the nodes share the throughput of one disk. In the Google cloud both are unnecessary. The access time to both the local root disk and the shared input data disk is the same, as they are both the same type of disks. The throughput is also unnecessary as the throughput of one disk is not shared by multiple nodes, as tested which gave the results stated in table 5.8. The processing times are higher for HDFS, as it has some overhead to keep track of the blocks. Also if a data block is not on the worker node's disk in HDFS, the block has to be transferred via the network from the node that has this block. This causes more time spent in communication, which slows down the processing.

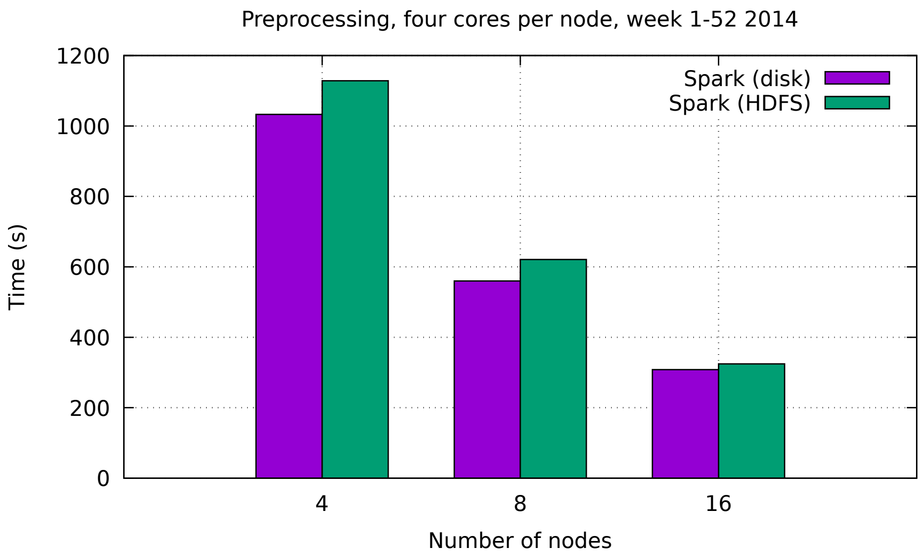


Figure 5.10: Pre-processing using HDFS as input source

5.6.4 Moving from HDD to SSD

Besides the standard hard drive disks, the Google cloud also offers SSD (solid state drive) disks. It is claimed that these disks have a much higher throughput and can handle more operations per second [37]. This is an interesting option to look at since we determined that the Flink implementation is limited by the disk speed.
Let us first starts with the root disk of each worker instance. This disk is used to spill the results when the memory is full. Replacing this with an SSD could therefore speed up the spilling and therefore the processing.

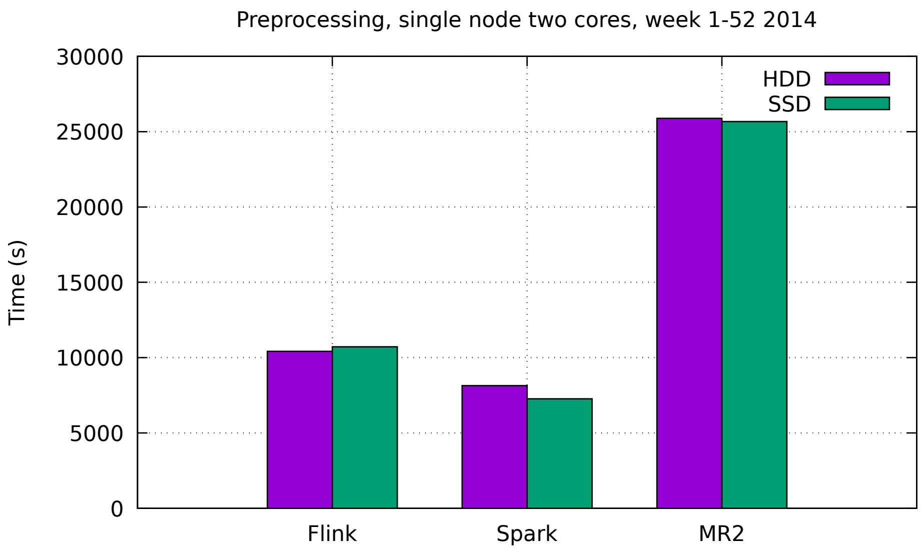


Figure 5.11: Pre-processing using an SSD as root disk

In figure 5.11 we can see the results when replacing the worker root disk with an SSD.

The test was executed on a two core machine to be sure that the Flink implementation is not, or in lesser ways, limited by the input disk.
The results dot not show the improvement as was expected. The Flink implementation seems to perform worse with an SSD. The MapReduce 2 implementation seems to perform slightly better. Both are however well within the 5 percent error margin equal to the HDD performance. The Spark performance seems to have improved the most.

To explain the results we again look at the disk I/O throughput. In figure 5.12 and table 5.9 the results of testing the disk I/O is given. The read speed is tested the same as in table 5.8 was used. The write speed is tested by writing 2 GB of data to a file using the *dd* command. The command is called as *dd if=/dev/zero of=test.dat bs=512 conv=fdatasync count=4000000*. After each test the caches are cleared to make sure the read and write speeds are disk bound and not memory bound.

The first observation we can make is that the write speed for HDD's and SSD's does not differ that much. The HDD's have a maximum write speed of 120 MB/s. The SSD's only have a write speed of 145 MB/s maximum. This can explain why the SSD root disk did not make much of a difference. The root disk is used for spilling, which is based on writing the data. Since the write speed did not increase that much, neither did the processing speed.

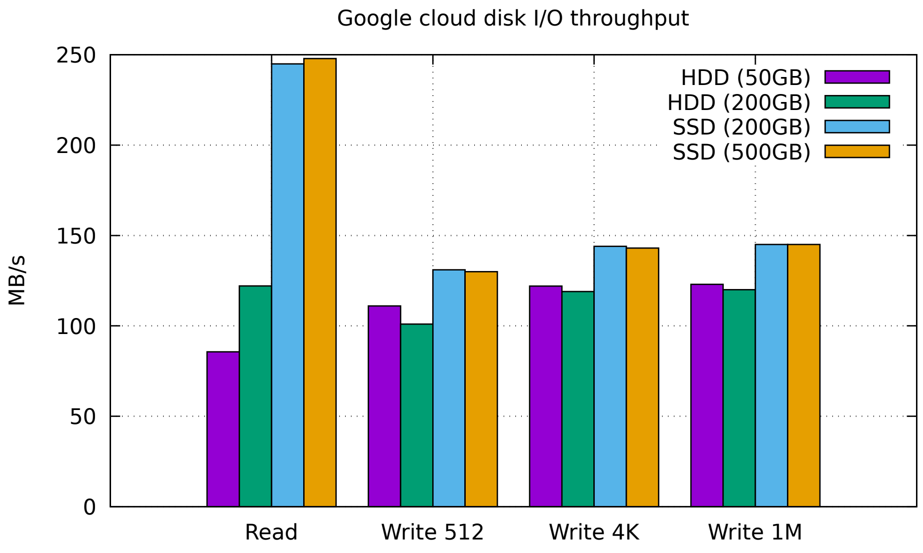


Figure 5.12: Google cloud disk throughput

Table 5.9: Google cloud disk throughput in MB/s

Speed	**HDD (200GB)**	**SSD (200GB)**	**SSD (500GB)**	**HDD (50GB)**
Read	122.06	244.92	247.84	85.63
Write 512	101	131	130	111
Write 4K	119	144	143	122
Write 1M	120	145	145	123

The second interesting observation we can make is the difference between the promised

throughput and the achieved throughput. The Google cloud documentation states that normal hard drive disks have a maximum sustained throughput related to the size of the disk. The maximum throughput is 0.12 MB/s per GB for both read and write speeds. The throughput is limited to a maximum of 180 MB/s for reading and 120 MB/s for writing [37]. For SSD's 0.48 MB/s per GB throughput with a maximum of 240 MB/s for both reading an writing is listed. Using these numbers the calculated maximum speeds for the disks tested should then be as stated in table 5.10.
Clearly those speeds do not even come close to the tested values.

Table 5.10: Claimed Google cloud maximum disk throughput in MB/s

Speed	**HDD (200GB)**	**SSD (200GB)**	**SSD (500GB)**	**HDD (50GB)**
Maximum read	24	96	240	6
Maximum write	24	96	240	6

What we do see is that the actual read throughput is much higher for the SSD's compared to the hard drives, where the write speed lacks. This sounds good for the input disk, as that only relies on the read speed. In figure 5.13 and table 5.11 we can see the results of replacing the input hard drive with an SSD. This test was done using 16 cores, to take the situation where the cores should fight the most for the disk.

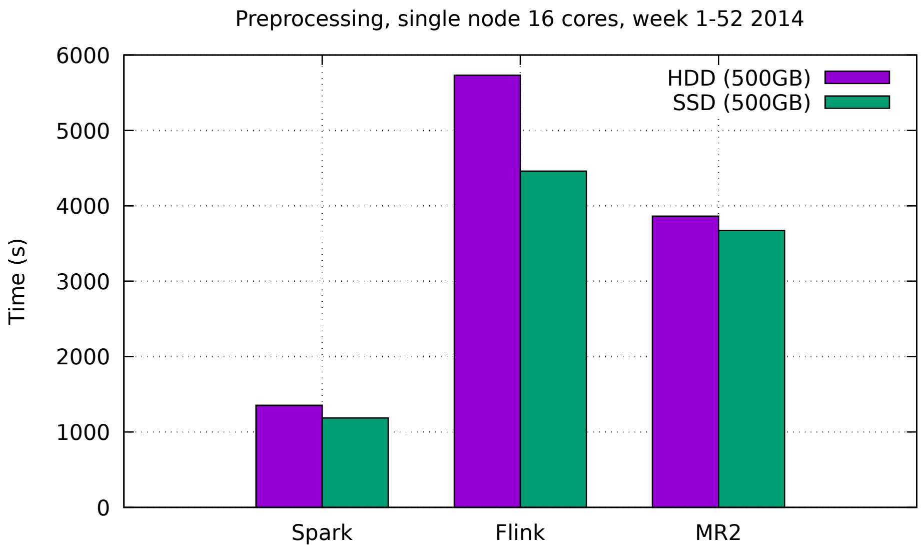


Figure 5.13: Pre-processing using an SSD as input source

Table 5.11: Pre-processing using an SSD as input source in seconds

	HDD (500GB)	**SSD (500GB)**
Spark	1354.09	1187.02
Flink	5732.85	4460.35
MR2	3863.56	3671.16

We can see that using an SSD as input disk does speed up the processing time for all implementations. The speed up for Spark and MapReduce 2 is minimal though. The real speedup can be seen for the Flink implementation. The processing time when using an SSD as input disk is more than 20 percent faster than the hard drive processing time. This again confirms the hypothesis that the Flink implementation was limited by the input disk speed in the core scaling test.

5.7 On demand

The tests of the on demand processing picks up where the pre-processing left. The testing for the on demand processing uses the intermediate files as generated by the pre-processing and stored in HDFS. This means that the temporal range is week 1 of 2014 to week 52 of 2014. The spatial range for which the trend analysis is done is set to the rectangle from 0 to 90 degrees for both latitude and longitude.

5.7.1 Scaling the number of cores

In figure 5.14 and table 5.12 we can see the results of scaling the amount of cores up for the on demand processing.

The first thing to notice is that the processing times are much lower than the pre-processing. Where the time of the pre-processing could be expressed in hours, the on demand processing should be expressed in seconds.
In the pre-processing tests the Spark platform was usually the fastest. From the figure it seems like the Flink implementation is faster for the on demand processing. It seems like MapReduce 2 is still the slowest, however the processing time decreases fast as the amount of cores increases. For Spark and Flink the processing time does decrease, but seems to flatten out.

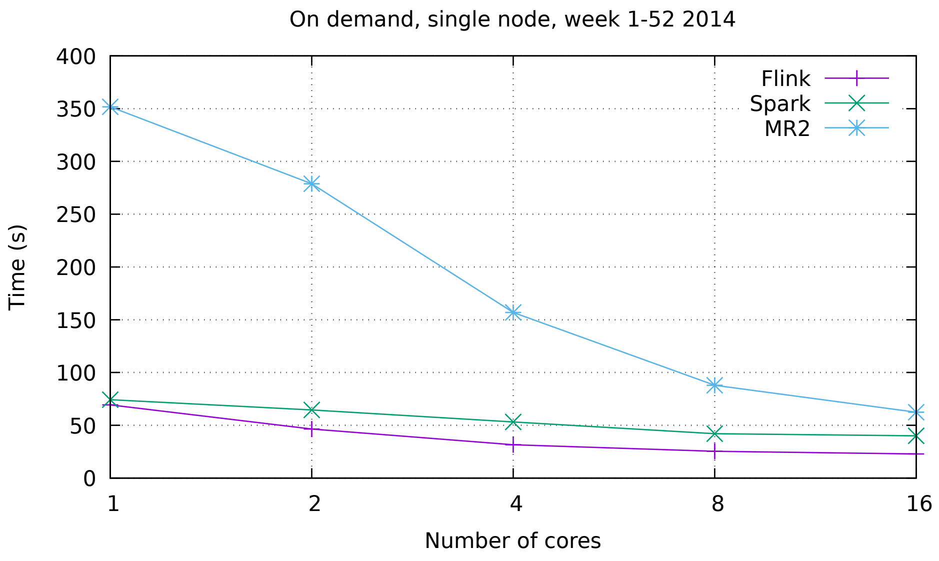


Figure 5.14: Measurements for different core counts

Table 5.12: Measurements for different core counts for on demand in seconds

#cores	Flink	Spark	MR2
1	69.33	74.31	351.80
2	46.54	64.59	278.84
4	31.64	53.22	156.86
8	25.38	42.13	87.96
16	22.91	40.09	62.45

In figure 5.15 and table 5.13 we see the speedup of the scaling of the cores. We see here again that the speedup decreases fast as the amount of cores increases. This can be explained by the structure of the application. The application reads the intermediate results, filters out for the spatial range, and averages the data to one measurement per week. Finally the trend analysis is done over those weekly measurements. The trend analysis is done as a sequential process, the remaining steps can be done in parallel. This means that the processing time cannot decrease in a linear way, as there is a sequential part which is not affected by the amount of cores. Another big impact in the processing time is the start up time. The processing platform needs some time to distribute the code, allocate the resources and start the application. The application master also needs some initialization time before any workers are started. This process takes about 20 seconds. This time is also not affected by the amount of cores.

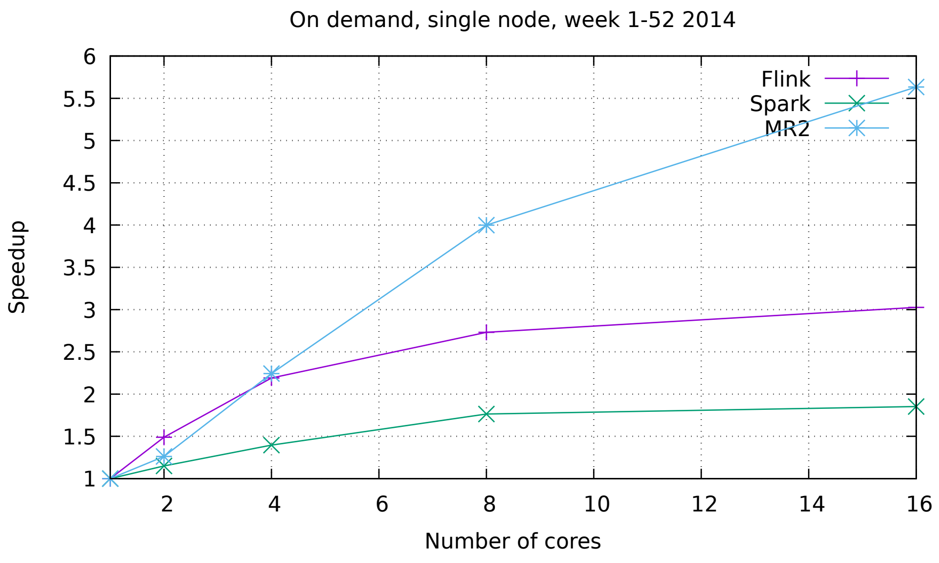


Figure 5.15: Speedup for different core counts

Table 5.13: Speedup for different core counts for on demand

#cores	Flink	Spark	MR2
1	1	1	1
2	1.49	1.15	1.26
4	2.19	1.40	2.24
8	2.73	1.76	4.00
16	3.03	1.85	5.63

5.7.2 Scaling the number of nodes

The same pattern appears for the scaling out with the amount of nodes in figure 5.16 and table 5.14. MapReduce 2 is again the slowest, but the processing time decreases fast. The Flink implementation is the fastest, and has a processing time that stays quite consistent. It is clear that the sequential parts of the trend analysis and the start up time takes the overhand in the processing time.

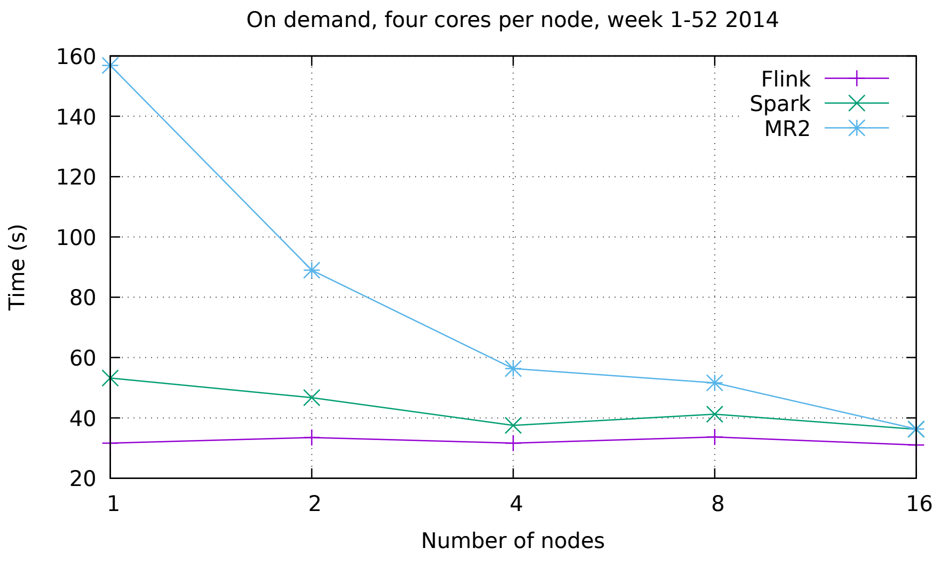


Figure 5.16: Measurements for different node counts

Table 5.14: Measurements for different node counts for on demand in seconds

#nodes	Flink	Spark	MR2
1	31.64	53.22	156.86
2	33.48	46.71	88.99
4	31.62	37.50	56.37
8	33.67	41.24	51.63
16	31.01	36.23	36.30

From the speedup of figure 5.17 and table 5.15 we can also see this. The speedup for MapReduce 2 goes up with the amount of cores, but is all but linear. The speedup for Spark and Flink seems quite constant within margin of error. This means that the Spark and Flink implementations do not scale at all for the on demand processing. The MapReduce 2 implementation does speed up with the amount of nodes, but scales very bad.

Table 5.15: Speedup for different node counts for on demand

#nodes	Flink	Spark	MR2
1	1	1	1
2	0.95	1.14	1.76
4	1.00	1.42	2.78
8	0.94	1.29	3.04
16	1.02	1.47	4.32

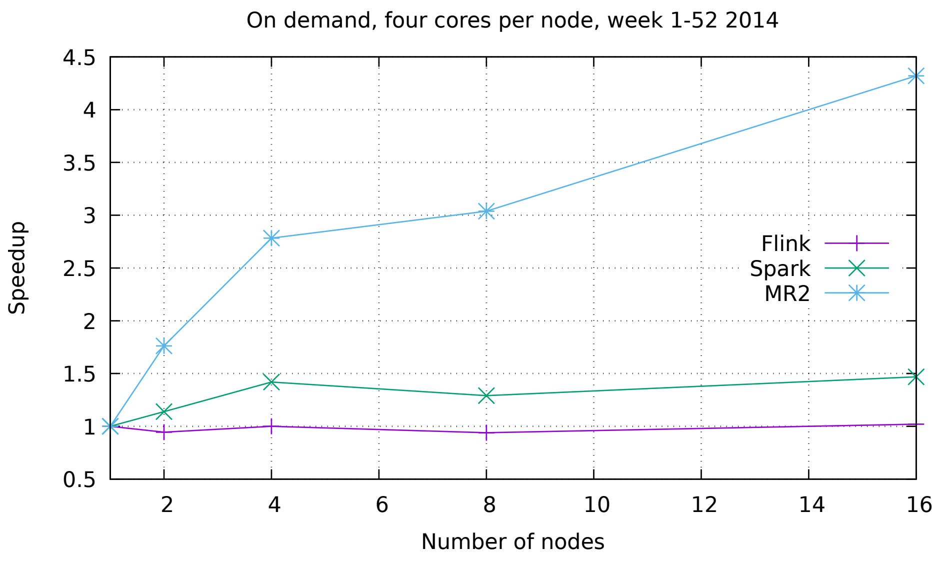
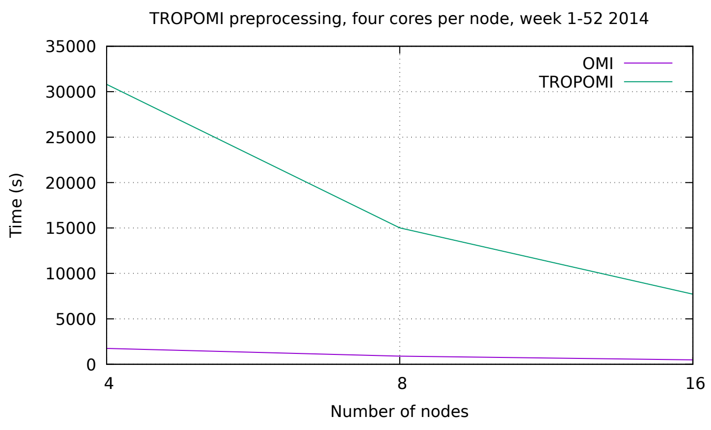


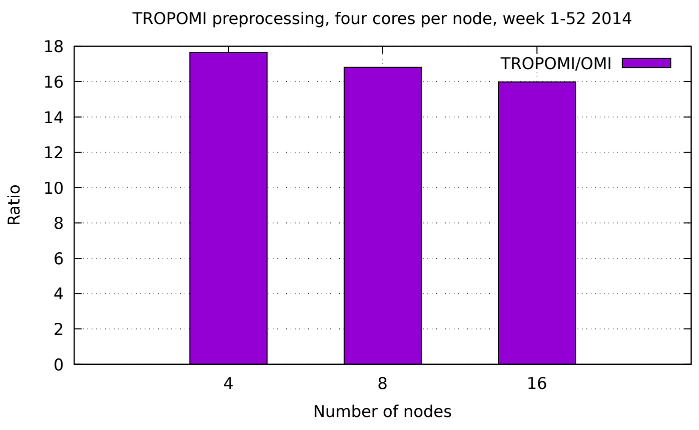
Figure 5.17: Speedup for different node counts

5.8 Scaling to TROPOMI

The final experiment is the pre-processing when scaling up of the input data to TROPOMI level. As the processing is expected to take a long time the test is only done for 4, 8 and 16 nodes. The test is done using the same range of input data as the tests using the OMI data. The amount of cores per node is set to 4, just like the OMI tests. Since the TROPOMI implementation is based on the Spark OMI implementation, we can compare the TROPOMI results to the OMI results based on the results of the Spark implementation.
The results can be seen in figure 5.18a and table 5.16. It is very clear that the processing time for the TROPOMI implementation is much longer than the Spark OMI version, as expected.



(a) Measurements for TROPOMI for different node counts



(b) Ratio of processing time TROPOMI versus OMI

Figure 5.18

In figure 5.18b we can see how the TROPOMI implementation performs compared to the OMI version. As expected the ratio comes out at around 16, which makes sense as we are using 16 times more data. The ratio is not a perfect 16 times, as the TROPOMI has a bit more overhead on the measurements. The TROPOMI version has to keep track of which of the 16 reading iterations a measurement was generated in. This causes a larger elements size in memory. As the memory size is limited the spilling phase is reached earlier that with the OMI version. This can also be seen by the ratio being slightly higher with less nodes. As more nodes are added, also more memory is added, the spilling phase is thus hit later, and the ratio becomes lower.

Table 5.16: Measurements for TROPOMI for different node counts in seconds

#nodes	**OMI**	**TROPOMI**
4	1746.11	30809.85
8	893.05	15007.84
16	483.25	7721.83

Chapter 6

Evaluation

Now that we have implemented and tested an application, we can move to the evaluation of the architecture. The goal of this evaluation is to discuss if the proposed architecture, which is based on the chosen software, matches the needs for the Downstream platform. This includes the qualitative aspects as well as the performance and scalability. As we have three versions of the architecture, based on Hadoop MapReduce 2, Apache Spark and Apache Flink, we can compare these three architectures.

6.1 Qualitative aspects

Let us start with some of the more qualitative aspects. The first aspect of the architecture is the needs for the applications. We have shown by implementing an application that all the processing platforms can be used for both pre-processing and on demand processing jobs. The YARN resource scheduler ensures than multiple applications can be used at the same time. If not enough resources are available, the applications are queued until the resources are available.

The portability aspect is also handled by YARN. As multiple processing platforms can run on top of YARN, we are not bound to the chosen processing platform.

The administration and a big part of the security aspect is handled by the interface. The actual requirements for the interface are dependent on the application. The second part of the security aspect is protection of the data from unauthorized access. This is done by encrypting the data. This encryption part is handled by HDFS, which has built in encryption [22]. Therefore we can use HDFS to store our input and intermediate data in an encrypted form.
When using the Google Cloud platform, as used for the testing environment, another form of encryption can be used. The disks in the Google cloud are by default encrypted on the server side [38]. This encryption is not noticeable when using the disks, and does not require any configuration. The keys are managed by the Google cloud, but does allow you to use your own keys. The downside is that you always have to share your keys with the Google cloud servers, as the encryption is always server side.

6.1.1 Reliability

Another aspect of the architecture is the reliability of the system. A mechanism for the reliability of the data is built into HDFS. HDFS replicates all data over the nodes. By default the replication is set to 3. This means that if a node in the cluster fails, at least two other nodes also have the same data.

The second part of the reliability, is the reliability of the processing platform. If a node fails, the system should be able to continue without that node. All three processing platforms have a form of reliability. The Hadoop MapReduce 2 system relies on YARN. As described in *Apache Hadoop yarn:Yet another resource negotiator* [39], YARN monitors all the nodes, and signals the application if any node fails. Hadoop MapReduce 2 picks up these signals and reschedules any tasks. Any work done by a map task on a failed node has to be re done, as the output is stored locally on the now failed node. The output of a reduce task is not stored locally, but on HDFS. Therefore the reduce tasks do not need to be executed again.

The Spark fault tolerance system also can use YARN to detect failed nodes. To detect missing data Spark uses a mechanism called Resilient distributed data sets [40]. A Resilient distributed data set, RDD, contains the data and the operation used to generate that data. When the operations are chained a directed acyclic graph of RDD's is created. When a node fails, the data it stored in memory and spilled on the local disk gets lost. Eventually when the next operation in the graph starts, the missing data is required. To restore this data the processing platform looks as the graph, it traces back into the graph to see which operation failed. This operation is repeated to regenerate the missing data. Note that the operation is only executed for the missing pieces of data. If that operation depends on data of a previous operation, which is not in memory anymore that operation also has to be repeated. This relies on the fact that the input data is also reliable, as the operations might have to be repeated up to the reading of the input data.

The Flink system is built for streaming. The batch processing is built on top of this streaming structure. The fault tolerance mechanism is therefore built for streaming. The fault tolerance system relies on a check pointing system [41]. Every so often a checkpoint is made of the state of the system. When a node goes down due to failure, it is detected by Flink. The system is reverted to the last checkpoint. Any tasks that were scheduled on the now absent node is done on a different node. The input that is also treated as a stream is also reset to the point of the checkpoint. This way the exact same data flows trough the processing, as it did originally after the checkpoint.

6.1.2 Creating applications

A important qualitative aspect is the creation of new applications. A part of this aspect is the programming language. The preferred language is Python. However when implementing the application using Python for all processing platforms, some problems where encountered, as described in section 4.8.1. Due to this Java was chosen as main programming language. This may be a problem for the application developers, as they are not used to program in Java.

Besides the programming language, the programming model may also be a hurdle. The programming model of the big data processing platforms is radically different from usual programming models. For developers that have not used this model, creating applications may become difficult. In chapter 4 we described the method used to implement the air quality application. This kind of method can be used as a guide line for developers to create other applications.

Related to the programming model is the programming API of the processing platforms. The proposal of the architecture contains the generalized processing library to smooth out the API for development of API's. This component however is not implemented yet.
When comparing the programming API's itself we can see the differences in the ease of implementation. The programming API of Hadoop MapReduce 2 is much more complicated than the Spark and Flink one. Due to the limited freedom of the MapReduce 2 API, the implementation of the air quality application pre-processing required to be split into two parts. Also the custom data types that are used to define the data, for example the combination of measurements, week and quality flags combination as described in section 4.8.3, requires a workaround in MapReduce 2. The exact same classes for those data types are used for Spark and Flink, however no workaround is required for those API's. Small problems like these may give developers, certainly those who are not experienced with the programming model, big problems. Therefore Spark and Flink are better for creating new applications.

6.1.3 Cost

The cost aspect in terms of money of running the architecture can be split up in two parts. The first part is the cost of the software used. In most cases software requires a license to be used. Often this license will cost a sum of money, either a single time payment or a monthly payment. A license for the software used, being Spark, Flink, HDFS, YARN, MapReduce 2 and Avro, is also required. All these pieces of software are published by the Apache Software Foundation. This means that all the used software is published under the open source Apache 2.0 license. The Apache 2.0 license is free for both personal and commercial use, so the cost of the software is non existent.

The second part of the cost of running the architecture is the hardware requirement. The architecture runs on top of hardware being either physical or virtual when clouds are used. This hardware costs money to run, and depends on the amount of servers and composition of those servers. Due to the set up of YARN and HDFS as described in section 3.3.1, a minimum of one head node and one worker node is required. The architecture is designed to scale, thus the performance scales with the amount of worker nodes. The performance is also influenced by the power of the individual nodes. This means that the cost is dependent on the required performance. If a better performance is required, a larger amount or more powerful nodes can be used, which will cost more money.

6.2 Scalability

One of the most important quantitative aspects, if not the most important one, is the scalability. The reason the new architecture makes use of big data software is their claimed ability to scale. This is required as the expected processing time will be too high if the architecture can not scale. We tested the scaling abilities for both scaling up in the amount of cores and scaling out in the amount of nodes in the cluster.

Of course these results are specifically for this implementation of the air quality application. Other applications may yield different results, as they for example might be more I/O bound. In such a case adding more cores might give a worse scaling than found in the results. It might also be that the application can not scale at all due to many data dependencies. For example the actual trend analysis process is a sequential algorithm, which is not parallelized in the implementation of the air quality application. The actual scalability results also depend on the quality of the implementation. If a developer makes a wrong choice, the system could be busy most of the execution time passing data around the nodes instead of processing. In such a case a sequential implementation can be better than a bad parallel implementation on big data processing platforms.

The results for the pre-processing show a near linear scaling for both the amount of cores and the amount of nodes for almost all the platforms. The exception being Flink in the scaling of the cores and MapReduce 2 in the scaling of the amount of nodes. This indicates that the architecture can indeed scale. An interesting result is the architecture scaling better with the amount of nodes than scaling the amount of cores. This is caused by the usage of the Google cloud, as it provides the disk resources in a way different than expected.
The on demand processing shows a different image. While the processing time of the execution of the on demand processing goes down with the amount of cores and nodes, the speedup is not even close to linear.

6.2.1 TROPOMI level data

The test with TROPOMI data shows if the architecture can not only scale with the amount of cores and nodes, but also with the amount of data. The TROPOMI data is about 16 times larger than the OMI data, the expected slowdown is therefore 16 times. From the results we can see that the slowdown is indeed about 16 times. The measured slowdown is a bit bigger than 16 due to the process of imitating the TROPOMI data with 16 times the OMI data. This shows us that the processing does scale linear with the amount of the data.

6.3 Performance

For the scalability aspect we looked at the speedups when increasing the cores and nodes. For the performance aspect we will look at the processing time itself. From the results we can see that the pre-processing time to process is indeed in the range of hours. The Spark implementation beats the other implementations on pure processing time. The processing time for the Spark implementation for a single year worth of data ranges from a little less than 3 hours for a single core to a bit over 5 minutes for the 16 node cluster. If we would extend that result to 10 years we would get a processing time of about one hour for the 16 node cluster. This certainly fulfills the need for the processing to complete within hours.

The MapReduce 2 implementation is by far the worst performing. The single core processing takes almost 10 hours on its own. The 16 node test took about 40 minutes to complete. If we scale that to 10 years worth of data we would get 6,67 hours of processing. While this processing time is within the range of hours, it is quite a bit longer than the Spark implementation. Note however that the any newly generated data does not have dependencies on the old data. This allows the processing to only process any newly generated data. This means that the actual data processing can be on a much smaller data set, and thus have a processing time that is well within an hour.

The on demand processing has some similarities with the pre-processing. The MapReduce 2 implementation is again the slowest by far. The difference being Flink is the fastest for the on demand processing. With enough nodes to process the data, the Flink implementation seems to average to about 30 seconds. The Spark implementation shows the same behaviour, but averages to about 40 seconds. The result of the processing time staying about the same with enough nodes, is caused by the start up overhead of the processing platform. It is unclear if the Spark implementation having a larger processing time than Flink is caused by being slower in the processing, or just having more start up overhead. The overhead leads to the on demand processing time being indeed in the range of seconds, just as mentioned in the user needs. However for a web based interface, a processing time of about 40 seconds is way too long. The user will click away long before the processing returns a result, thinking the page has become unresponsive.

Chapter 7

Future work

The research in this thesis has raised a few new questions which can be pursued in future research.

Firstly this research only covered software like HDFS, Avro, MapReduce 2, Spark and Flink. These pieces of software were chosen in the architecture based on some aspects deemed important. Maybe some other software that may be less widely known, or does not fit the aspects precisely may end up being better. These other software platforms may be a very strong candidate on some aspect, but being just less fit in another aspect, and thus not considered.
This leads to the question if the chosen programming model is indeed the correct one. The three processing platforms, MapReduce 2, Spark and Flink were chosen partially because they are quite popular. Those platforms use (a derivative of) the map reduce model to parallelize the processing of the data. Maybe some other programming model, which can be used to parallelize the processing of the data, may fit the applications even better. A couple of example software platforms which have a different programming model are given in section 3.2.2.

A question related to the programming model is the granularity of the processing, as covered in section 4.5. In the implementation of the air quality application a granularity of one measurement is used, as it is assumed to be the easiest to implement. A future line of research could be to find out the effect of increasing the granularity. Does it make the implementation harder, and is the performance impacted by said granularity?

In the chapter describing the architecture we mentioned the generalized geo processing library in section 3.2.5. This library was not implemented due to time constraints. This library however is an interesting research topic. It requires the collection of requirements and ideas, which are then translated to an API. The second part is figuring out if implementing this geo library is actually possible. The processing model of the map reduce processing platforms is very different from general programming. It may not be possible, or feasible due to extreme overheads, to make a mapping from the API to the map reduce model.

The final question that arose is the solution to the on demand processing time. The on demand processing is severely limited by the start up time of the processing frameworks. This causes the processing time to be way too high to be usable as a service. A solution could be looking in to keeping the processes that execute the processing alive after a job has finished. When this is achieved, one could also look in to keeping the intermediate results in memory to speed up the on demand processing even more. To do that the problem of the separated memory spaces, as described in section 5.4 has to be solved.

Chapter 8

Conclusion

In this research we have assessed the usage of big data software for remote sensing applications on the Downstream platform. We have answered the question how big data software can be used to enhance the Downstream platform by creating an architecture built on this software. The software chosen is HDFS and Apache Avro for storage and Hadoop MapReduce 2, Apache Spark and Apache Flink being the candidates for the processing platform. The processing platforms were compared by implementing the air quality application on them. This application is split in a time consuming pre-processing step, and a lightweight on demand step.

Testing this application, by using the Google cloud for the (virtual) machines, showed an interesting result. For the pre-processing Flink struggles when adding more cores due to lacking disk performance. Hadoop MapReduce 2, being the slowest, starts to struggle when more nodes are added. This is caused by the tasks finishing to fast, and the start up overhead of starting tasks in MapReduce 2 being too high. Spark performs without struggle when adding more cores and nodes.

For the on demand processing no platform seems to struggle when more cores or nodes are added. The MapReduce 2 platform is again the slowest of the three. The performance of Flink and Spark seems to be limited by a start up overhead, which really harms the scalability of the application. This also makes any service based on the processing slow, and therefore possibly unusable.

From these results, being that Spark does not seem to struggle in the pre-processing and the comparison on the other aspects, Apache Spark seems to be the best processing platform. We can conclude that the usage of big data software as described in the architecture can indeed be used for applications that do pre-processing like processing. For applications that provide services, just like the on demand processing of the air quality application, the architecture as of now may not be the best solution yet. To make the architecture usable for these class of applications, the problem with the start up overhead has to be fixed first.

Bibliography

[1] Manyika James, Chui Michael, Brown Brad, Bughin Jacques, D Richard, R Charles, and HB Angela. Big data: The next frontier for innovation, competition, and productivity. *The McKinsey Global Institute*, 2011.

[2] Doug Laney. 3D data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[3] Erwin Goor, Jeroen Dries, and Dirk Daems. Mission exploitation platform proba-v. In Soille and Marchetti [42], pages 50–51.

[4] Christoph Reck, Gina Campuzano, Klaus Dengler, Torsten Heinen, and Mario Winkler. German copernicus data access and exploitation collaborative infrastructure. In Soille and Marchetti [42], pages 52–55.

[5] Norman Fomferra. *Cal/Val and User Services - Calvalus Final report.* Brockmann Consult GmbH, October 2011. `http://www.brockmann-consult.de/calvalus/pub/docs/Calvalus-Final_Report-Public-1.0-20111031.pdf`.

[6] Dabin Christophe, Holliman Mark, Melchior Martin, Belikov Andrey, and Hoar John. Euclid: Orchestrating the software development and the scientific data production in a map reduce paradigm. In Soille and Marchetti [42], pages 5–8.

[7] Fabio Pasian, John Hoar, Marc Sauvage, Christophe Dabin, Maurice Poncet, and Oriana Mansutti. Science ground segment for the esa euclid mission. In *SPIE Astronomical Telescopes+ Instrumentation*, pages 845104–845104. International Society for Optics and Photonics, 2012.

[8] Sort FAQ. `http://sortbenchmark.org/FAQ-2016.html`, May 2016.

[9] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. GraySort on Apache Spark by Databricks. November 2014.

[10] Thomas Graves. GraySort and MinuteSort at Yahoo on Hadoop 0.23. May 2013.

[11] Machines Algorithms and People Lab UC Berkeley. Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`, Februari 2014.

[12] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[13] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. Bdgs: A scalable big data generator suite in big data benchmarking. In *Workshop on Big Data Benchmarks*, pages 138–154. Springer, 2013.

[14] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S Pérez-Hernández. Spark versus flink: Understanding performance in big data analytics frameworks. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 433–442. IEEE, 2016.

[15] The Apache Software Foundation. Welcome to apache hadoop. `http://hadoop.apache.org/#Download+Hadoop`, March 2016. Acessed 10 August 2016.

[16] The Apache Software Foundation. Spark release 2.0.0. `http://spark.apache.org/releases/spark-release-2-0-0.html`, July 2016. Acessed 10 August 2016.

[17] The Apache Software Foundation. Apache flink: Downloads. `http://flink.apache.org/downloads.html`, Augustus 2016. Acessed 10 August 2016.

[18] Sagar Nikam. Projects - other than hadoop! `https://azadparinda.wordpress.com/2013/10/11/projects-other-than-hadoop/`, October 2011. Acessed 5 August 2016.

[19] James Golick. What does ”scalable database” mean? `http://jamesgolick.com/2010/3/30/what-does-scalable-database-mean.html`, March 2010. Acessed 5 August 2016.

[20] Matt Allen. Relational databases are not designed for scale. `http://www.marklogic.com/blog/relational-databases-scale/`, November 2014. Acessed 5 August 2016.

[21] The Apache Software Foundation. HDFS permissions guide. `https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html`, January 2016. Acessed 5 August 2016.

[22] The Apache Software Foundation. Transparent encryption in HDFS. `https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html`, January 2016. Acessed 5 August 2016.

[23] Abhishek Jain. Parquet file format. `http://bigdata.devcodenote.com/2015/04/parquet-file-format.html`, April 2015. Acessed 5 August 2016.

[24] Pascal S. de Kloe. jvm-serializers. `https://github.com/eishay/jvm-serializers/wiki`, July 2016.

[25] Martin Kleppman. Schema evolution in Avro, Protocol Buffers and Thrift. `https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html`, December 2012.

[26] DataXu. Three reasons why apache avro data serialization is a good choice for OpenRTB. `http://blog.cloudera.com/blog/2011/05/three-reasons-why-apache-avro-data-serialization-is-a-good-choice-for-openrtb/`, May 2011. Acessed 5 August 2016.

[27] Michael Wetzel, Tamir Melamed, Mark Vayman, and Denny Lee. Using avro with hdinsight on azure at 343 industries. `https://dennyglee.com/2013/03/12/using-avro-with-hdinsight-on-azure-at-343-industries/`, March 2013. Acessed 5 August 2016.

[28] The Apache Software Foundation. Apache avro releases. `http://avro.apache.org/releases.html`, April 2016. Acessed 10 August 2016.

[29] The HDF Group. High Level Introduction to HDF5. `https://support.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf`, September 2016.

[30] Kevin O'Dell. How-to: Select the right hardware for your new hadoop cluster. `http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/`, August 2013. Acessed 23 August 2016.

[31] Pieternel F Levelt, Gijsbertus HJ van den Oord, Marcel R Dobber, Anssi Malkki, Huib Visser, Johan de Vries, Piet Stammes, Jens OV Lundell, and Heikki Saari. The ozone monitoring instrument. *IEEE Transactions on geoscience and remote sensing*, 44(5):1093–1101, 2006.

[32] National Aeronautics and Space Administration. Aura. `https://aura.gsfc.nasa.gov/about.html`. Acessed 10 August 2016.

[33] National Aeronautics and Space Administration. OMI User's Guide. `https://disc.gsfc.nasa.gov/Aura/data-holdings/additional/documentation/README.OMI_DUG.pdf`. Acessed 10 August 2016.

[34] Unidata, UCAR Community Programs. NetCDF-Java Library. `http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/documentation.htm`, May 2015.

[35] The Apache Software Foundation. Apache avro 1.8.1 hadoop mapreduce guide. `http://avro.apache.org/docs/1.8.1/mr.html`, May 2016.

[36] Salvatore Valente and Karel Zak. kill(1) - Linux manual page. `http://man7.org/linux/man-pages/man1/kill.1.html`, July 2014.

[37] Google. Optimizing Persistent Disk and Local SSD Performance. `https://cloud.google.com/compute/docs/disks/performance`, January 2017.

[38] Google. Managing Data Encryption — Cloud Storage Documentation — Google Cloud Platform. `https://cloud.google.com/storage/docs/encryption`, January 2017.

[39] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. 2012.

[41] Apache Software Foundation. Apache Flink 1.1.3 Documentation: Data Streaming Fault Tolerance. `https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/stream_checkpointing.html`. Acessed 31 January 2017.

[42] Pierre Soille and Pier Giorgio Marchetti, editors. *Proc. of the 2016 conference on Big Data from Space (BiDS16)*. Publications Office of the European Union, 2016.