Master System and Network Engineering

University of Amsterdam

System and Network Engineering

Large-scale Netflow Information Management

Adrien Raulot
adrien.raulot@os3.nl

Shahrukh Zaidi
shahrukh.zaidi@os3.nl

February 11, 2018

Supervisor: Wim Biemolt (SURFnet)

Abstract

In order to better respond to network incidents, NetFlow data analysis has become an important traffic monitoring technique. NfDump is a widely used tool to analyse NetFlow data. However, this single-threaded solution starts to become inadequate as networks become larger along with the amount of data to analyse. This research project focuses on finding a technique to analyse NetFlow data in a more time-efficient manner. After investigating the limitations of NfDump, our study and evaluation of different Big Data analysis techniques has resulted in a new approach using Apache Hadoop and Apache Spark. A proof of concept using these technologies has been implemented in order to compare this solution to NfDump. The measurements indicate a clear performance improvement in terms of time-efficiency using Hadoop and Spark when analysing large amounts of data. We conclude that the combination of Apache Hadoop and Apache Spark is a valuable choice as a distributed solution to efficiently analyse NetFlow data.

Contents

1 Introduction **7**
- 1.1 Ethical considerations . . . 7

2 NetFlow **9**
- 2.1 What is NetFlow? . . . 9
- 2.2 NetFlow analysis . . . 10

3 NetFlow Analysis techniques **11**
- 3.1 Current setup . . . 11
  - 3.1.1 Overview of current setup . . . 11
  - 3.1.2 Limitations of current setup . . . 12
- 3.2 Netflow analysis techniques . . . 12
  - 3.2.1 ELK Stack . . . 12
  - 3.2.2 Hadoop . . . 13
  - 3.2.3 Apache Spark . . . 15
- 3.3 Choice for analysis technique . . . 15
  - 3.3.1 Hadoop 2.0 & PySpark . . . 15
  - 3.3.2 Parquet . . . 16

4 Experiments **17**
- 4.1 Environment . . . 17
- 4.2 Implementation . . . 17
  - 4.2.1 Preparing the input data . . . 17
  - 4.2.2 Conversion . . . 18
  - 4.2.3 Querying the data . . . 19
- 4.3 Spark optimizations and tuning . . . 20
- 4.4 Test queries . . . 20

5 Results **23**
- 5.1 Query execution time . . . 23
  - 5.1.1 Retrieve all flows containing a specific IP address . . . 23
  - 5.1.2 Retrieve all flows with a byte count larger than 100MB . . . 24
  - 5.1.3 List the top 10 of Telnet connections with only the SYN flag set in the IP header ordered by the number of bits per second . . . 24
  - 5.1.4 List the top 10 of IP addresses receiving the largest amount of traffic . . . 25
  - 5.1.5 Retrieve all flows with only the SYN flag set in the IP header . . . 26
- 5.2 Accuracy . . . 26

6 Discussion **27**
- 6.1 Further optimizations . . . 28
- 6.2 Conclusion . . . 28

Bibliography **31**

CHAPTER 1

Introduction

NetFlow data provides valuable information about network users, applications and network usage. This enables organisations to use network analysis tooling to see what is happening on their networks and perform network forensics. Details of network traffic statistics, e.g. source and destination IP addresses or TCP/UDP port numbers, are stored and used for, for example, incident analysis. Organisations that process large amounts of network data need to be able to analyse these large amounts of data in an acceptable time limit. Therefore, the choice of tooling can be of great significance in order to meet the time requirements.

The current NetFlow analysis tools used by the SURFcert incident and response team are becoming insufficient in terms of time-efficiency for large-scale NetFlow analysis. Querying NetFlow data over long time frames is time-consuming and inconvenient when the team has time-constraints to analyse the network traffic. The objective of this study is to examine whether other techniques exist, that deliver better performance in terms of data analysis speed and efficiency than the current NetFlow data analysis system used by SURFnet. Our main research question is:

Which data analysis technique could be used in order to analyse the current SURFnet NetFlow data in a more time-efficient manner?

In this study we will limit ourselves to improving the querying speed of the NetFlow data. In order to get an answer to our main research question, we will have to answer a number of sub-questions:

- What makes the current setup time-inefficient?
- What techniques could be of use to overcome this performance bottleneck?
- Would these techniques be future-proof when dealing with a higher sampling rate?

The remainder of this study is structured as follows: Chapter 2 explains the NetFlow protocol in more detail and describes its usefulness. Chapter 3 gives us an overview of the current setup used by SURFnet to analyse NetFlow data and discusses its limitations. Related work is discussed in order to find techniques that could potentially overcome these limitations and the chosen techniques are described. Chapter 4 provides an overview of the experiments and describes the implementation in detail. Chapter 5 presents the results of our experiments. Finally, chapter 6 discusses our findings and possible optimizations, and concludes the research.

1.1 Ethical considerations

Network data can contain sensitive information about users of the network. To test the proof of concept implementation of the NetFlow analysis setup, we have worked with the NetFlow data of SURFnet. However, the data used for these testing purposes is less sensitive and will not contain traffic of end users but only data from services within the network. We will use the data for the purposes of this research only. Finally, any NetFlow data included in this report will be obfuscated.

CHAPTER 2

NetFlow

2.1 What is NetFlow?

NetFlow refers to a traffic monitoring technology originally developed by Cisco. The protocol allows for monitoring IP network traffic passing through routers or switches without having to inspect every packet. Since Cisco made the technology publicly available, the protocol has become a widely used technique for network analysis[19].

The Internet Engineering Task Force (IETF) has defined a *flow* as follows[26]: "a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties." These common properties may consist of, amongst others, the source and destination IP addresses and ports of a packet, the transport layer protocol and other packet header fields. These properties are extracted by a NetFlow enabled router and used to determine whether an IP packet belongs to an existing flow or whether a new flow is observed. The flows are then stored for further analysis. Figure 2.1 gives an schematic overview of this process:

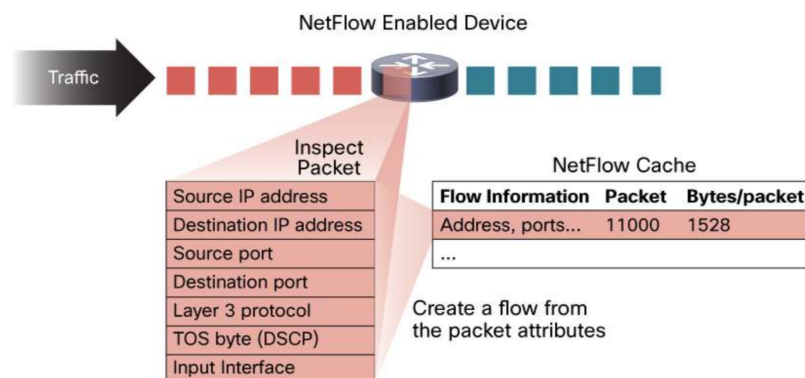

Figure 2.1: Schematic overview of the NetFlow export process[17]

There are several versions of NetFlow being used, with NetFlow versions 5 and 9 being the most popular[24]. NetFlow v5 was the first raw storage representation for network flow data to see wide adoption. It is a unidirectional protocol based on a fixed-length binary record format, with a fixed set of fields[20]. The NetFlow v5 data stream consists of a number of packets. Each packet has a header followed by a number of records. These records contain start and end timestamps, source and destination addresses and ports, protocol, type-of-service, TCP flags in the flow, input and output interface, source and destination autonomous system (AS) number and source and destination prefix mask length. As a result of the fixed structure of NetFlow v5 data, this version only supports the export of IPv4 flows and 16-bit AS numbers. This limitation led to the introduction of NetFlow v9[19]. This version allowed for more extensibility and flexibility by abandoning the fixed record format for a template-based system where the record format can be manually defined. This template-based system enabled the integration

of MPLS, IPv6 and other user defined records. In 2004, the IETF started standardizing a flow export protocol. NetFlow v9 of Cisco was chosen as the basis for this standardization[23]. IPFIX is a template-based, record-oriented, binary export protocol, with the *message* as its basic unit of data transfer[20]. A message contains a header and one or more *sets*. IPFIX achieves flexibility by defining two types of sets: template sets and data sets. A template set contains template records, a data record contains data records. Each data record has a reference to a template that describes the data within the set.

There are a number of important differences between NetFlow export and regular packet capture. Flow export only examines packet headers whereas regular packet capture methods consider individual packet payloads. Therefore, flow export is considered less privacy sensitive[19]. Because only the packet headers are considered, significant data reduction can be achieved. This significant reduction of data also leads to less consumption of computational resources for analysis in comparison with regular packet inspection[24]. As processing, storing and analysing each packet in large-scale networks is not feasible, analysis of NetFlow has become crucial.

2.2 NetFlow analysis

Netflow data is collected in order to discover useful information by using statistics or other sophisticated approaches[24]. Hofstede et al.[19] have defined three main application areas of NetFlow analysis: flow analysis and reporting, threat detection, and performance monitoring.

Flow analysis is used to filter flow data and obtain general statistics and information about the network. For example, it allows network administrators to get insight into the *top-talkers* in the network. These are the users or services within the network exchanging the largest amount of traffic. Another example would be the distribution of traffic throughout the day or week. Reporting is used to set alerts based on unwanted situations, e.g. when an unusual number of connections is being generated by a host.

Threat detection can be divided into two types of uses[19]. Flow data can be used to determine which hosts have been communicating with each other. Combined with an IP blacklist this could help identify traffic from suspicious hosts. Secondly, threat detection can be applied by utilizing the definition of a flow for analysing certain types of threats. NetFlow data can be analysed for specific traffic pattern to identify network scans, Distributed Denial of Service (DDoS) attacks, worm spreading, and botnets.

Finally, NetFlow analysis can be used for performance monitoring by observing the status of services running on the network. Reported metrics may include the Round-Trip-Time (RTT), delay, jitter, response time, packet loss, and bandwidth usage.

CHAPTER 3

NetFlow Analysis techniques

3.1 Current setup

3.1.1 Overview of current setup

To analyse their network traffic, the SURFcert incident response team currently uses the NfDump[8] and NfSen[9] tool set. The NfDump tool is used to collect and analyse NetFlow data on the command line and consists of several components. The two most important of these components are `nfcapd` and `nfdump`.

`nfcapd` is the NetFlow capture daemon of the NfDump tool set. One `nfcapd` process is responsible for each NetFlow stream. SURFnet currently has two routers that export NetFlow data and, therefore, two daemons running on the network. The daemons read the NetFlow data from the network and store the data into time-sliced files. The sampling rate of the data affects the number of packets that are considered for NetFlow processing and is configurable. The goal of using sampling is to reduce the processing requirements for analysis of the NetFlow data. Two types of sampling can be distinguished[19]. Systematic sampling decides deterministically whether a packet is selected for flow analysis, for example by selecting every Nth packet. In random sampling, however, packets are considered in accordance to a random process. The SURFcert team currently uses systematic sampling with a sampling rate of 1 out of 100 and a time frame of 5 minutes of NetFlow data per file.

`nfdump` is the NetFlow display and analysing program of the NfDump tool set. It is responsible for reading the files stored by `nfcapd` and processing the contained flows. It is used to analyse the NetFlow data using its powerful filter syntax by enabling network administrators to display the raw NetFlow data and flow element statistics.

Figure 3.1 gives an example overview of the NfDump tool set:

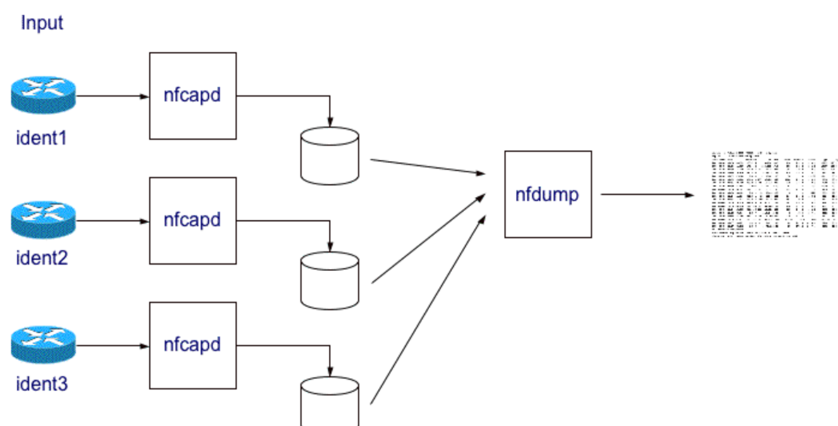

Figure 3.1: Schematic overview of the NfDump tool set[8]

By storing all data to disk before analysis, NfDump separates the process of storing and analysing data. This allows for analysis of NetFlow data from the past and tracking interesting traffic patterns. The analysis can be done for a single 5 minutes time frame file. However, in order to find interesting patterns, it is more useful to look at larger time spans. NfDump allows this by concatenating multiple files for a single run.

NfSen is graphical web based front-end for the NfDump NetFlow tools[9]. It uses `nfdump` as back-end tool to visualize the NetFlow patterns of the current network situation, graph specific profiles or specific time windows[18]. It allows for easy navigation through the NetFlow data and enables network administrators to set alerts based on various conditions, create queries using custom filters and create and manage profiles to retrieve data that matches certain conditions.

3.1.2 Limitations of current setup

As provider of Internet and ICT facilities to students, lecturers and scientists of Dutch education and research institutions, SURFnet has to deal with large amounts of data on their network. Even though the NfDump and NfSen tools have proven to be reliable techniques for network analysis, the increasing amount of data that is being generated has made these tools insufficient in terms of time-efficiency. Tian[27] has described the following limitations when using NfDump for large-scale Netflow analysis:

- **Inefficient file-based store**: as NfDump typically stores NetFlow data in separate files for every 5 minutes time frame, data for a long time span is contained in a large number of files.
- **Very slow processing speed**: each file is read line by line by NfDump from the beginning. Therefore, analysis of large amounts of NetFlow data takes a lot of time. Searching for a specific data entity in a large volume of data is time-inefficient when the data is spread over many files as every single line of these file will need to be read.
- **Limited analysis methods**: as network situations are becoming more and more complex, new analysis approaches are required that allow for NetFlow data analysis.

As shown in figure 3.1, the `nfdump` process to analyse NetFlow data operates on a single server environment. When handling large amounts of data, this may not scale well in terms of memory, processing speed, and data storage capacity. SURFnet has made efforts in the past to optimize the NfDump source code by applying parallelization techniques. This optimization led to a 10% time-efficiency gain. However, this did not prove to be sufficient.

3.2 Netflow analysis techniques

After studying the current setup and finding out its limitations as described in the previous section, it appears that a distributed approach would be better in terms of scalability in order to achieve better time performance. A distributed approach would speed up the processing speed as more hardware can be used to perform the analysis. Moreover, additional hardware would allow for the separate files to be read in parallel.

Different distributed techniques exist that allow for large scale data analysis. In the following sections we will elaborate on a few different computing techniques that may be of use when analysing NetFlow data.

3.2.1 ELK Stack

One of the techniques used recently as an alternative to NfDump is the ELK Stack[3]. ELK (now evolved to the Elastic Stack) stands for Elasticsearch, Logstash and Kibana, which are three open-source projects. Logstash is able to gather large amounts of data from multiple simultaneous sources, transform those data and send them to a stash like Elasticsearch for search and analytics. Kibana is a front-end that lets the user vizualize the data with charts and graphs in Elasticsearch.

Figure 3.2 gives an overview of the ELK Stack:

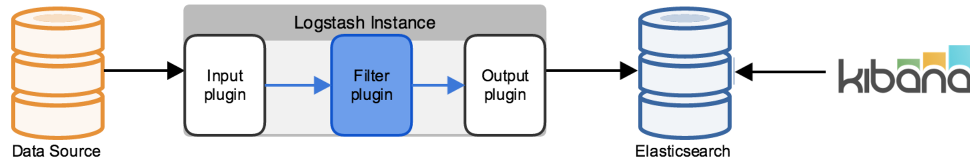

Figure 3.2: Schematic overview of the ELK Stack[3]

In the recent past, Tian[27] has studied the use of the ELK stack as a NetFlow analysis technique and concluded that ELK appears to be a promising method to analyse NetFlow data. The study has found Logstash to be a flexible tool for collecting, organizing and storing the NetFlow data and ElasticSearch appears to be able execute simple searches quickly. However, the author argues that ELK may have some limitations when analysing complicated network behavior. Moreover, the experiments in this study are carried out on a limited data set and the performance on large data sets is not evaluated.

To assess whether ELK could be a viable option for NetFlow analysis at SURFnet, a research project involving this technique has been conducted in the past by SURFnet. The experiments were run on a cluster composed of 9 data nodes used for data related operations such as search and aggregations, and 3 master nodes that control the cluster. The NetFlow codec plugin for Logstash[7] has been used to decode NetFlow data. Even though this technique seemed promising at first, it appeared to be unadapted for the amount of data SURFnet has to handle. Elasticsearch can handle both structured and full text data and is very fast to perform aggregations. However, this project has shown that Elasticsearch lacks in processing speed when it comes to executing complex queries on large amounts of NetFlow data. Logstash has accentuated this problem of performance by processing flows at a maximum rate of 15,000 flows per second per instance only, which makes it too slow for the amount of data of SURFnet. For these reasons, the ELK Stack has not been retained as a viable option to analyse SURFnet's NetFlow data.

3.2.2 Hadoop

The Hadoop software[1] is a framework developed in Java by the Apache Software Foundation that allows for processing of a large data sets in a distributed manner. The Hadoop project includes modules such as the Hadoop Distributed Filesystem (HDFS) for storage, Hadoop YARN (Yet Another Resource Negotiator) for job scheduling and cluster resource management or Hadoop MapReduce for parallel processing of large data sets. It has been designed to scale up to thousands of machines and be very time-efficient by offering local computation and storage. A large number of Hadoop-related projects[6] has been developed such as data warehouses, serialization systems, databases or computing engines and many others, allowing the Hadoop software to be more accessible and used as a foundation for a wide variety of projects.

Figure 3.3 gives an overview of Hadoop 1.0:

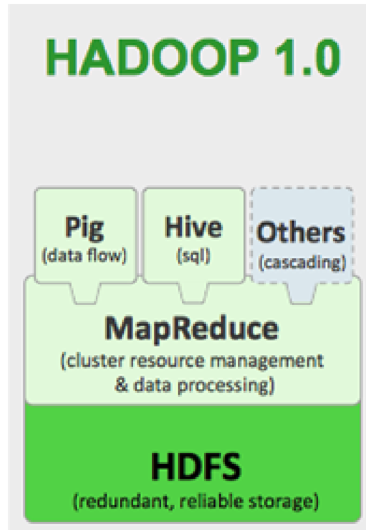

Figure 3.3: Schematic overview of Hadoop 1.0[5]

In Hadoop 1.0, Hadoop MapReduce was responsible for managing the cluster resources and processing batch data. Since Hadoop 2.0, Hadoop YARN has been added as a new layer between HDFS and MapReduce, taking care of cluster resource management and solving many of the scaling, availability and resources utilization issues of Hadoop 1.0. In Hadoop 2.0, Hadoop MapReduce only performs batch data processing and Hadoop has become a multi-purpose platform, capable of running batch as well as interactive and streaming applications.

Figure 3.4 gives an overview of Hadoop 2.0:

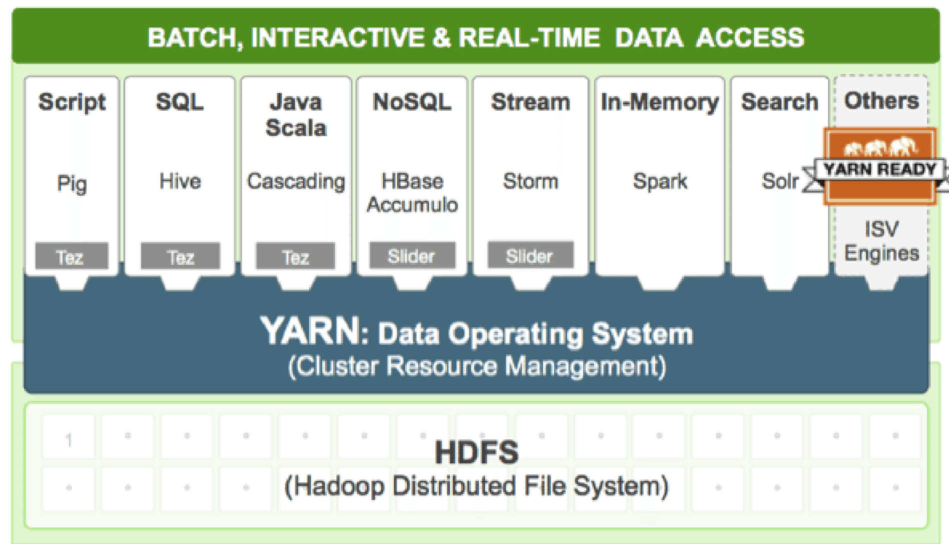

Figure 3.4: Schematic overview of Hadoop 2.0[4]

Each of the Hadoop-related projects[6] has been designed to answer different needs and therefore changes the way data is processed.

In recent years, a number of studies have been conducted on Hadoop as a network traffic analysing technique. Lee and Lee[22] have presented a Hadoop-based NetFlow analysis system using MapReduce capable of analysing multi-terabytes of Internet traffic in a scalable manner. The experiments are performed on two Hadoop testbeds, with a 30-node and 200-node cluster respectively. The performance results show high throughputs for the data analysis tasks with higher observed throughputs when in-

creasing the number of nodes. Comparing the setup with a single-server analysis tool, up to 20.3 times increased throughput is achieved for the 30-node cluster. However, several hardware-related performance bottlenecks are identified as well. One of these is related to the memory requirements. Map tasks of the analysis jobs write intermediate results to disk. These results are pulled by reducers using remote procedure calls. With high volumes of network data the hard disk capacity needs to be large enough to store the intermediate map results.

A similar study[21] that has utilized the MapReduce framework to analyse NetFlow data has shown that Hadoop appears to be scalable in large-scale networks. Experiments with four Hadoop data nodes showed that the flow computation time can be significantly improved by 72% compared with typical flow analysis tools.

3.2.3 Apache Spark

Apache Spark[2], originally developed at the University of California, Berkeley's AMPLab, and now maintained by the Apache Software Foundation, is an open-source cluster-computing framework. Apache Spark is intended to be a very powerful computing engine for Big Data processing and is much faster than Hadoop MapReduce by exploiting in-memory computing and other optimizations. It can access diverse data sources including HDFS and comes with built-in modules for streaming, SQL, machine learning and graph processing. The ease of use of Apache Spark APIs, providing bindings for the Java, Scala, Python and R programming languages, is one more reason for Spark to be a valuable option when operating on large data sets.

There are relatively few studies in the area of NetFlow analysis using Apache Spark. Čermák et al.[16] set up a series a of experiments to benchmark the performance of different stream processing systems to analyse NetFlow data in real-time. The benchmark is used to compare Spark, Samza, and Storm, currently the most widely used distributed stream processing systems, to determine their suitability for flow analysis. It is important to note that distributed stream processing is different from traditional batch processing as data is not persistently stored on disk or memory, but only exists as a stream of messages. In this study, the authors have selected a set of operations inspired by common security analysis methods of flow data. These operations include filtering, counting, aggregation, and top N statistics of the NetFlow data. The performance of these data processing systems is assessed by measuring the throughput when executing these operations. The benchmark results show that Apache Spark has a high enough flow throughput to be able to process at least 500,000 flows per second using 16 or 32 processor cores. The authors argue that the system is probably capable of even higher throughput using a more space-efficient data format than the one used in the study (JSON). It is therefore considered to be perfectly suitable for large-scale flow data processing.

3.3 Choice for analysis technique

3.3.1 Hadoop 2.0 & PySpark

In order to decide upon a certain set of techniques to analyse NetFlow data, we first need to identify the needs of the SURFcert team. The time taken to perform queries on a large amount of data is currently too high. Moreover, a possible increase of the amount of data in the future as well as a preferred higher sampling rate of NetFlow data have to be taken into consideration. With regard to the scope of this project, our goal is to make the querying on data more time-efficient. As discussed in the previous section, prior studies have shown that Hadoop 2.0 is very efficient for Big Data analysis. Apache Spark has been shown to achieve high throughputs when analysing NetFlow data. Therefore, we have decided that the combination of Hadoop 2.0 and Spark is the most promising set of techniques to reach the goal of this project. With YARN and HDFS bringing important measures of scalability and reliability to Hadoop and Spark as a powerful computing engine for data processing, it should be possible to reach high availability, scalability and reliability, which would most likely result in a significantly reduced querying time. Finally, the Python API of Spark, PySpark, is our choice of API. One can argue that Python for Spark is slower than Scala in terms of performance. However, Python comes with its flexibility, its ease of use, its large number of libraries and its robustness which are numerous reasons for Python to be a preferred choice.

3.3.2 Parquet

Choosing the right storage format is fundamental when dealing with large amounts of data, even more when fast access to the data is an important requirement. As described in section 2.1, the template-based NetFlow data version 9 may include lots of different fields. Querying NetFlow data mostly consists of working on a subset of columns, because only a few columns are usually needed in order to obtain the desired information. Apache Parquet, an open-source data store for the Hadoop ecosystem, is our choice of storage format. Parquet is column-oriented, meaning that it stores the data by column rather than row, which by design, provides faster access to data when only a subset of column rather than the entire record is needed. In their research [15], Bittorf et al. describe Parquet as a format optimized for large data blocks (tens, hundreds, thousands of megabytes) with built-in support for nested data and offering both high compression and scan efficiency.

The compression algorithm used when storing data is also of importance. Although the CPU overhead has to be taken in consideration, using data compression reduces the space requirements on disk and improves I/O efficiency. When running a very I/O intensive job on a large Hadoop cluster, I/O perfomance quickly becomes the bottleneck rather than CPU. Various data compression formats may be used with Hadoop such as LZO, BZip2, Gzip and Snappy. LZO is a popular compression format used with Hadoop that offers very fast decompression and allows for the user to adjust the balance between compression ratio and compression speed, without affecting the speed of decompression. Previous research[15] has established that among all the compression formats, Snappy is the best choice when it comes to compression and decompression speed. Snappy has been developed at Google and designed to be fast and not CPU intensive, which makes it a valuable tool in a Hadoop cluster. According to prior studies[25], Snappy may be faster than LZO but the compression ratio of Snappy seems to a be a bit less efficient.

CHAPTER 4

Experiments

4.1 Environment

In order to be able to deal efficiently with the large amount of NetFlow data SURFnet has, we have run our experiments on a Hadoop cluster composed of several nodes with decent computing resources. We make use of the Big Data services provided by SURFsara, offering multiple frameworks and public data sets to researchers for Big Data analysis. A large Hadoop cluster is used, composed of:

- $\sim$ 100 nodes
- $\sim$ 600 cores
- $\sim$ 4TB of memory
- $\sim$ 2PB of storage
- Apache Hadoop 2.7.2
- Apache Spark 2.1.1

The cluster specifications may seem impressive, but in practice we only used part of the cluster resources, with YARN dynamically taking care of the resource management. However, certain properties, discussed in section 4.3, may manually be ajusted in order to define the amount of resources attributed to our jobs. Our NfDump setup where the same queries are processed by NfDump is as such:

- 1x Dell PowerEdge R230
- Intel Xeon CPU E3-1240L v5 @ 2.10GHz
- 4 cores
- 16GB of RAM
- $\sim$ 200GB of SSD storage
- NfDump v1.6.12

4.2 Implementation

4.2.1 Preparing the input data

The first step is to import the NetFlow data into HDFS, the Hadoop Distributed File System. Our dataset is composed of 61GB of version 9 NetFlow data split into 81 files which are the result of the NetFlow capture daemon (`nfcapd`) capturing 7 hours of traffic. Although we found a repository[13] allowing NetFlow binary data files to be imported into Spark and read from Spark SQL, we could not make use of it because this library does not support version 9 NetFlow data files at the moment. As the NetFlow binary files can not be directly imported into Spark, we first have to convert the data to CSV. Export to this format is supported by NfDump using the following command:

```
$ nfdump -r nfcapd.201801011245 -o extended -o csv -q > csv/nfcapd
    .201801011245.csv
```

This command reads the file *nfcapd.201801011245* containing NetFlow data, and outputs each flow in CSV format to the standard output. In order to have more details about the flow, the *-o extended* option is passed to the command. Finally, the header line and statistics from NfDump are suppressed using *-q* and the standard output is redirected to a CSV file.

Once all data is converted to CSV, the CSV files are uploaded to HDFS using HDFS CLI, the HDFS command line client. The command to upload the files is as follow:

```
$ hdfs dfs -put *.csv /path/on/hdfs/test_data/csv/
```

Entering the command above uploads all the local CSV files to the path */path/on/hdfs/test_data/csv/* on HDFS.

4.2.2 Conversion

Converting all NetFlow data from binary to CSV format results in a total of more than 580GB of CSV data. We have written a Spark Python script to load and convert this data to Parquet data for efficient querying. Algorithm 1 contains the pseudo-code of this script:

Algorithm 1 Converter

1:	*df_csv* ← load('*.csv')	▷ loads all data from CSV files into new DataFrame 'df_csv'
2:	*df_csv*.select('ts', 'td', ...)	▷ selects only desired columns from 'df'
3:	*df* ← *df_csv*.castToInt('td')	▷ casts 'df_csv' columns to desired types
4:	*df*.write('data.parquet')	▷ save 'df' data to Parquet file
5:	exit	

This already demonstrates the ease of use of Spark and its Python API PySpark, where only a few simple lines of code suffice to create a powerful Spark job. This distributed job, when executed by YARN, is split into many tasks executed on the different nodes of the cluster in a multi-threaded environment. The resource manager of Hadoop comes with a Web UI that provides information about the different jobs, the resource allocations and access to logs and metrics. Figure 4.1 gives an overview of the ResourceManager Web UI showing the different stages of the converting job being executed:

Stages for All Jobs

Active Stages: 1
Completed Stages: 2

Active Stages (1)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	load at NativeMethodAccessorImpl.java:0 (kill) +details	2018/01/30 16:24:43	21 s	1/5049	4.2 GB			

Completed Stages (2)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	load at NativeMethodAccessorImpl.java:0 +details	2018/01/30 16:24:39	0.8 s	1/1	64.0 KB			
0	load at NativeMethodAccessorImpl.java:0 +details	2018/01/30 16:24:31	6 s	99/99				

Figure 4.1: Overview of the converting Spark job as shown on the Hadoop ResourceManager Web UI

The converting job described in algorithm 1 results in the creation of 5035 Snappy Parquet files of various sizes, from a few kilobytes to hundreds of megabytes each. Each of these files is the result of a task and has been compressed using Snappy (described in section 3.3.2), the compression library used by SURFsara's Hadoop cluster.

4.2.3 Querying the data

Although the approach to query data using Spark is very different from NfDump, it is important to retrieve the same information. We have created a Spark job for the querying process using Python, described by algorithm 2:

Algorithm 2 Querier

1: *df* ← load('*.parquet') ▷ loads all data from Parquet files into new DataFrame 'df'
2: *df*.createTempView('nf') ▷ creates a temporary view 'nf' from DataFrame 'df'
3: *query* ← 'SELECT ts, sa, da FROM nf' ▷ Spark SQL query
4: *res* ← execute(*query*) ▷ new DataFrame 'res'
5: dropTempView('nf')
6: *res*.cache() ▷ caches 'res' into memory
7: print *res*.count() ▷ displays number of records
8: print *res*.show() ▷ displays part of the records in a table
9: dropFromCache(*res*)
10: exit

The script starts by loading the previously generated Parquet files containing our NetFlow data into a DataFrame. The Spark Documentation[10] defines a Spark DataFrame as "conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood". A temporary view (line 2) has to be created in order for Spark to be able to run our SQL query later on. The data can be queried by creating a string containing our SQL query using the typical SQL syntax and executing it afterwards. The temporary view created beforehand is used in the query as the equivalent of a table in SQL. The result of the query execution will be returned as a DataFrame, containing all the matching records. As NfDump displays the total number of matching flows and their content, we included the records display and count in our script as well.

4.3 Spark optimizations and tuning

One of the key factors in time-efficiency when using Spark is caching. Spark is capable of storing data into memory in order to process operations much faster. We noticed that *count()* and *show()* (line 7 and 8 of algorithm 2) were the most time-consuming instructions, with the process of loading and querying the data being very fast. This is because Spark is lazy in a sense that any selecting of filtering instructions that might be specified earlier in a job will not be executed until Spark has to, typically when calling *show()* or *count()*. When using data caching, as can be seen on line 7, we noticed a net decrease of the job execution time.

Another factor of optimization is the management of the different Spark configuration properties. Those properties can be statically managed (i.e., hard-coded in Spark configuration files) or dynamically managed by specifying certain flags in command line when submitting a Spark job. The command we used to submit our Querier Spark job is:

```
$ spark-submit --master yarn --deploy-mode cluster querier.py
```

The first option, *–master*, is used to specify the master URL of the cluster to submit the job to. In our case the value is 'yarn' as we are connecting to a YARN cluster, with the URL being defined in the Spark configuration files. The second option, *–deploy-mode*, is set to 'cluster' as we want to work with all the nodes of our cluster. Other flags[11] may be used in order to modify the way our job is executed on the cluster. By default, Apache Spark will automatically attribute values to these flags or use the ones defined in the Spark configuration files. However, it may be interesting to consider changing some of those properties values dynamically. The command below includes all the flags we used in order to optimize the execution of our Querier job:

```
$ spark-submit --master yarn --deploy-mode cluster --num-executors 16 --
    executor-cores 4 --executor-memory 18G querier.py
```

Executors are processes on a cluster node that run computation and store data for our application. By explicitly specifying the number of executors to 16, we are able to distribute the computing work for our job more fairly taken the scale and the specifications of our cluster into consideration. We have also set the number of cores per executor to 4 and the amount of memory for each executor to 18GB to allow more resources to each executor, thus changing the execution behavior of our job and being more time-efficient. There are relatively few studies on methods to determine whether it is best to set the flags to specific values depending on the cluster specifications and the Spark job, in order to get the best performance out of Spark. Using the ResourceManager Web UI of Hadoop, it is possible to get information about the garbage collection time of each executor which may be an indicator of the resource distribution quality. Finally, submitting a job several times using different reasonable flag values and comparing its execution time seem to be the best approach at the moment.

4.4 Test queries

In order to analyse the performance of our setup using Hadoop and Spark, we will compare the execution times of a set of different queries on both the current setup with NfDump and the distributed implementation. The queries that are used for this comparison consist of a number of common queries that may be executed by the SURFcert team to analyse the Internet traffic on their network for a certain time frame. The data that is used for these performance tests consists of 7 hours of NetFlow data with a total of nearly 2.8 billion flows. For each individual query 5 different time spans are used to measure the execution time of the query: 5 minutes, 30 minutes, 1 hour, 3.5 hours, and 7 hours. The queries executed to test the performance are described below:

- **Retrieve all flows containing a specific IP address**
  Analyse the traffic to or from a specific IP address. An example would be when a malicious IP address is detected on the network. The query allows to get insight into the behavior of the malicious user.
- **Retrieve all flows with a byte count larger than 100MB**
  Analyse all IP addresses that are either sending or receiving more than 100MBs of data in a single flow.

- **List the top 10 of Telnet connections with only the SYN flag set in the IP header ordered by the number of bits per second**
  Analyse the network to find all IP addresses starting a Telnet connection (port 23 or 992) ordering the found records by bits per second. This query may be used to identify possible TCP SYN scanners on the SURFnet network.

- **List the top 10 of IP addresses receiving the largest amount of traffic**
  Analyse whether a certain IP address is receiving an unusually high amount of data. This query may be used to find a possible DDoS victim.

- **Retrieve all flows with only the SYN flag set in the IP header**
  Get an overview of all IP addresses initiating a TCP connection.

CHAPTER 5

Results

5.1 Query execution time

5.1.1 Retrieve all flows containing a specific IP address

The first experiment that we have conducted covers the retrieval of all flows that contain a specific IP address, either as source or as destination address. The measured execution times are shown in figure 5.1 below:

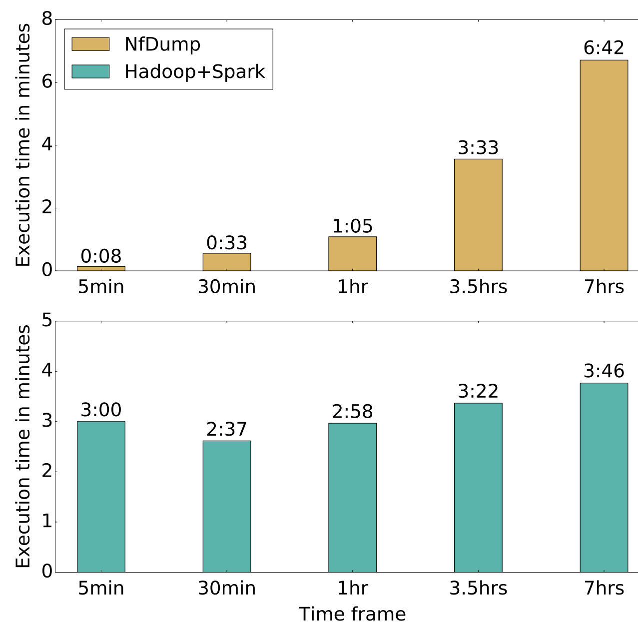

Figure 5.1: Execution time of retrieving all flows containing a specific IP address.

It is apparent from this figure that for short time frames NfDump is able to retrieve the requested data more quickly. Up until a 1-hour timespan, NfDump is significantly faster to find the flows containing the specified IP address. A possible explanation for this might be that Hadoop requires some time to start up. The tasks need to be defined and, along with the data, distributed over the different nodes. The Java Virtual Machine (JVM) also requires time to initialize certain data structures. However, once the time frame is extended to cover larger time frames, the benefits of the scalability of Hadoop are revealed. Where the execution time of the query on NfDump roughly doubles when increasing the timespan by a factor of two, the execution time on the Hadoop implementation does not appear to be affected as significantly. Another rather surprising result is the lower execution time when the time frame is extended from 5 to 30 minutes. It is difficult to explain this result, but it might be related to a better distribution

of jobs and data for this specific amount of NetFlow data.

5.1.2 Retrieve all flows with a byte count larger than 100MB

The second set of experiments aimed to measure the data retrieval time when retrieving all flows with a total byte count of larger than 100MB. Figure 5.2 below presents the results obtained for these experiments:

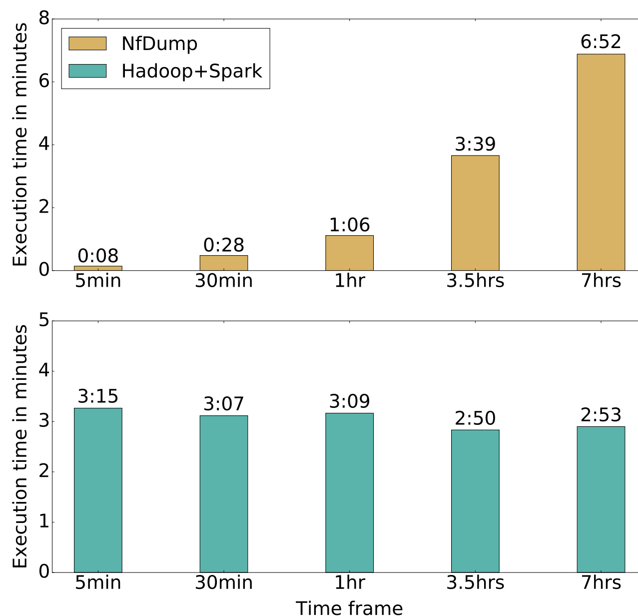

Figure 5.2: Execution time of retrieving all flows with a byte count larger than 100MB.

The results are similar to those obtained in the first experiment. The computation time of NfDump appears to increase linearly as the time frame is extended, whereas that of Hadoop does not quickly build up. On the contrary, for this experiment the execution time on Hadoop improves for larger time frames.

5.1.3 List the top 10 of Telnet connections with only the SYN flag set in the IP header ordered by the number of bits per second

The next experiment conducted retrieves all Telnet connections with only the SYN flag set in the IP header. The results are ordered by the number of bits per second and the top 10 flows are returned. The queries for these experiments require some more complicated filtering and aggregation. The results of this set of experiments are shown in figure 5.3:

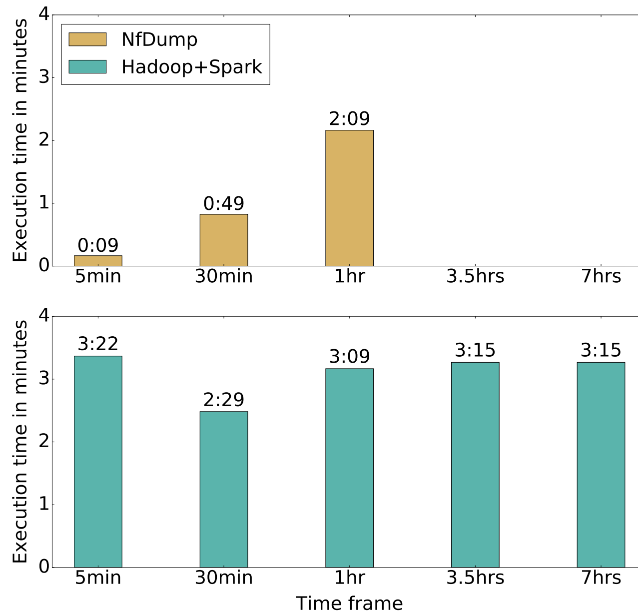

Figure 5.3: Execution time of listing the top 10 of Telnet connections with only the SYN flag set ordered by the number of bits per second.

The single most surprising observation to emerge from this figure is the lack of data for NfDump for the 3.5 and 7 hours time frames. For a reason, unknown to us, NfDump was not able to execute the query for this experiment for a timespan larger than 1 hour and simply stopped the execution. A possible explanation for this might be that NfDump is not able to execute such complex queries on a large amount of data.

5.1.4 List the top 10 of IP addresses receiving the largest amount of traffic

In this experiment we have measured the computation time of the queries that are used to retrieve the IP addresses that receive the largest amount of traffic for a certain time frame. The top 10 of these addresses is returned. Figure 5.4 shows the outcome of the experiments:

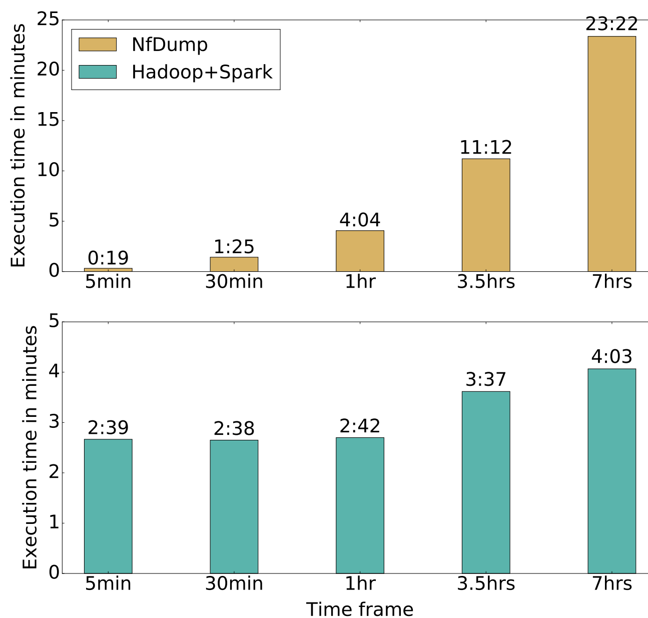

Figure 5.4: Execution time of listing the top 10 IP addresses receiving the largest amount of traffic.

This figure shows dramatic differences between the execution times on NfDump and Hadoop. For 7 hours of NetFlow data, NfDump takes almost 6 times longer to return the requested top 10 IP addresses. This factor is expected to increase even more for increasing amounts of data as the computation time for NfDump has appeared to increase linearly with larger amounts of data. The computation for Hadoop increases as well. However, the increase seems to be much more gradual.

5.1.5 Retrieve all flows with only the SYN flag set in the IP header

The last set of experiments covers the retrieval of all flows that have only the SYN flag set in the IP header. The results are shown in figure 5.5 below:

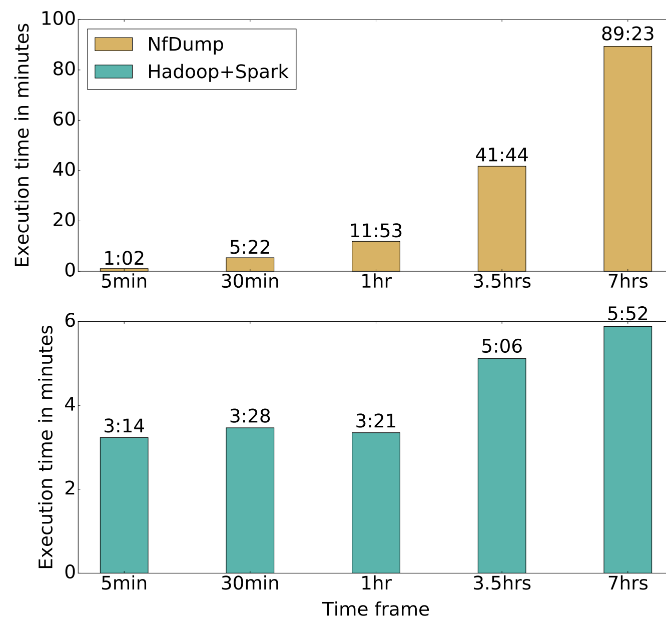

Figure 5.5: Execution time of retrieving all flows with only the SYN flag set in the IP header.

For this experiment the computation times appear to diverge even more dramatically. For a relatively small time frame of 30 minutes, NfDump already takes longer to retrieve the requested flows. This difference becomes even more significant when the time frame is extended. For a 7-hour timespan we observe a huge discrepancy with the execution on NfDump taking almost 15 times longer. It is difficult to explain the reason for this excessive computation time, but it might be related to the way NfDump compares the TCP flags. However, to answer this question a thorough inspection of the NfDump source code would be required.

5.2 Accuracy

To make sure that the results these experiments have yielded are valid, we have investigated whether the number of flows returned for both NfDump and Hadoop are similar. We found that for some of the executed queries there is a slight deviation in the number of resulting flows. However, the observed difference was not significant and, therefore, we do not believe that it has an major effect on the results of these experiments.

CHAPTER 6
Discussion

The present study was designed to determine whether a more time-efficient method exists in order to analyse large-scale NetFlow data. In order to answer this question we have first identified the performance bottlenecks of the current analysing setup of SURFnet. The single-threaded processing approach of NfDump combined with the inefficient file-based store emerged as the most important limitation of the current situation. After finding out these limitations, we decided that a distributed approach could accelerate the processing speed significantly. The combination of Apache Hadoop and Apache Spark was chosen as a promising solution. To assess whether these technologies can indeed be a valuable NetFlow analysis method, a proof-of-concept is built and the performance is compared to the current setup using NfDump.

One of the significant findings to emerge from this study is that for relatively short timespans NfDump is able to retrieve the queried data more quickly. This can most likely be explained by the fact that the Hadoop ecosystem requires some time to start up, define the jobs, initialize certain data structures, and distribute these jobs along with the data over the different nodes. However, when the timespan is increased and more NetFlow data needs to be considered, the benefits of the scalability of Hadoop and the high throughput of Spark become apparent. The results of this study show that the execution time of queries on NfDump appear to increase linearly as the amount of data that has to be processed increases. In contrast, Hadoop is able to handle large amounts of NetFlow data much more easily and the computation times do not quickly build up. The differences between the execution times are highlighted even more when more complex queries are performed, where even for short time frames Hadoop and Spark are able to retrieve the relevant flows significantly faster.

The second aim of this study was to determine whether the chosen analysis technique would be future-proof when dealing with a higher sampling rate. The current sampling rate of 1 out of 100 results in a flow export of roughly 70,000 flows per second. As discussed in section 3.2.3, previous research has shown that Apache Spark is capable of processing up to 500,000 flows per second with opportunities for further optimization to achieve even higher throughput. Therefore, we believe that the combination of Hadoop and Spark is able to handle higher sampling rates. However, to determine whether an ideal sampling rate of 1 out of 1 is feasible, further studies need to be carried out in order to assess the maximal throughput that can be achieved using these two technologies.

As discussed in the introduction of this work, the scope of this study was limited to finding a solution to optimize the querying speed of large-scale NetFlow data. However, to extract interesting patterns from the queried data, visualization of the flows is an important aspect of NetFlow analysis. Even though we believe that integrating the two techniques used in this study with the visualization capabilities of NfSen might be challenging, Python offers its own powerful visualization libraries. An example would be the `plotly` package that allows for visualization of Spark data frames. Another possibility would be to convert the data frames into the `pandas` data structure as this library offers a wide range of data visualization capabilities. However, further studies need to be carried out in order to analyse the most efficient way to visualize the NetFlow data.

6.1 Further optimizations

In subsection 4.3, we have seen that our Spark jobs required to be manually optimized in order to have a better execution in terms of performance. However, more can be done and areas such as garbage collection, data serialization, memory usage and caching would need to be further investigated in terms of optimization, as explained in the Apache Spark Documentation[12]. The Spark APIs offers several programming language options, each of them having its own benefits. Subsection 3.3.1 describes the reasons behind our choice of using the Python API of Spark for the implementation of this project. In the future, the implementation could be further expanded and involve more data processing. In that case, it may be important to consider using Scala[14] as it is known to be much faster when significant processing logic is involved.

6.2 Conclusion

The aim of the present research was to examine what data processing technique could be of use in order to analyse the large amount of SURFnet's NetFlow data in a more time-efficient manner. After identifying the bottlenecks of the current analysing method and reviewing the relevant literature, the combination of Hadoop and Apache Spark appeared to be a promising solution. To determine whether the combination of these technologies could indeed be a valuable NetFlow analysis method, a proof-of-concept implementation was designed. The performance of this implementation was then evaluated by comparing the execution times of a set of different queries on both the current setup with NfDump and the distributed implementation. This study has shown that this distributed approach can indeed be a time-efficient solution to achieve faster query times when analysing large-scale NetFlow data. The scalability of Hadoop combined with the high throughput offered by Apache Spark make these techniques a valuable NetFlow analysis method. However, the SURFcert team typically executes queries on relatively short timespans. As queries on these timespans are in most cases more time-efficient on NfDump, the team will have to decide whether the current tool set can be used alongside Hadoop and Spark to combine the advantages of both systems.

Acknowledgments

We would first like to thank our supervisor Wim Biemolt from SURFnet. We thank Matthijs Moed from SURFsara for his assistance with the Hadoop cluster and technologies, Roland Van Rijswijk-Deij from SURFnet for his guidance on the different Big Data analysis techniques and Marijke Kaat for pointing us in the right direction at the start of the project.

Bibliography

[1] Apache Hadoop. https://hadoop.apache.org/.

[2] Apache Spark. https://spark.apache.org/.

[3] ELK Stack. https://www.elastic.co/elk-stack.

[4] Getting Started with Hadoop 2.0..! http://www.xoomtrainings.com/blog/getting-started-with-hadoop-2-0.

[5] Hadoop 2.0 (YARN) Framework - The Gateway to Easier Programming for Hadoop Users. https://www.dezyre.com/article/hadoop-2-0-yarn-framework-the-gateway-to-easier-programming-for-hadoop-users/84.

[6] The Hadoop Ecosystem Table. https://hadoopecosystemtable.github.io/.

[7] Netflow codec plugin. https://www.elastic.co/guide/en/logstash/current/plugins-codecs-netflow.html.

[8] NfDump. http://nfdump.sourceforge.net/.

[9] NfSen - Netflow Sensor. http://nfsen.sourceforge.net/.

[10] Spark Documentation. https://spark.apache.org/docs/latest/sql-programming-guide.html.

[11] Spark Documentation - Submitting applications. https://spark.apache.org/docs/latest/submitting-applications.html.

[12] Spark Documentation - Tuning Spark. https://spark.apache.org/docs/latest/tuning.html.

[13] spark-netflow. https://github.com/sadikovi/spark-netflow.

[14] The Scala Programming language. https://www.scala-lang.org/.

[15] M. Bittorf, T. Bobrovytsky, C. C. A. C. J. Erickson, M. G. D. Hecht, M. J. I. J. L. Kuff, D. K. A. Leblang, N. L. I. P. H. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[16] M. Čermák, D. Tovarňák, M. Laštovička, and P. Čeleda. A performance benchmark for netflow data analysis on distributed stream processing systems. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 919–924. IEEE, 2016.

[17] I. Cisco. NetFlow. Introduction to Cisco IOS NetFlow C a technical overview, 2007.

[18] P. Haag. Watch your flows with NfSen and NfDump. In *50th RIPE Meeting*, 2005.

[19] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[20] C. M. Inacio and B. Trammell. Yaf: yet another flowmeter. In *Proceedings of LISA10: 24th Large Installation System Administration Conference*, page 107, 2010.

[21] Y. Lee, W. Kang, and H. Son. An internet traffic analysis method with mapreduce. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 357–361. IEEE, 2010.

[22] Y. Lee and Y. Lfee. Toward scalable internet traffic measurement and analysis with hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2013.

[23] S. Leinen. Evaluation of candidate protocols for ip flow information export (ipfix). 2004.

[24] B. Li, J. Springer, G. Bebis, and M. H. Gunes. A survey of network flow applications. *Journal of Network and Computer Applications*, 36(2):567–581, 2013.

[25] P. Raichand. A short survey of data compression techniques for column oriented databases. *International Journal of Global Research in Computer Science (UGC Approved Journal)*, 4(7):43–46, 2013.

[26] G. Sadasivan. Architecture for ip flow information export. *Architecture*, 2009.

[27] Z. Tian. Management of large scale NetFlow data by distributed systems. Master's thesis, NTNU, 2016.