Analysing Common Crawl - Efficient and Cost-Effective Processing of Large-Scale Data

Lukasz Domanski

MInf Project (Part 2) Report

Master of Informatics

School of Informatics

University of Edinburgh

2020

Abstract

The Web is a popular area of research; however, investigating the Web's properties presents a number of technical challenges as processing terabytes (or even petabytes) of data is required if the aim is to make inferences about the Web as a whole. This report investigates the most efficient ways of analysing Common Crawl in the context of workloads performing simple computation on data of this magnitude. The suspicion was that in some scenarios the performance of Apache Hadoop can be suboptimal and significant runtime reduction is possible if a different solution is used to process the data.

This hypothesis was confirmed by comparing a number of solutions, including Apache Spark and simple bash utilities, in a variety of settings and scenarios including a single-node local setup, a multi-node cluster hosted in the cloud, processing data available locally, streaming data stored remotely, as well as using specialised hardware offering a high level of parallelism. However, as migrating to a different Big Data framework is not always straightforward, a tool called Wee Common Crawl Utility, which allows reusing existing Hadoop Streaming code while achieving a significant runtime reduction, was developed by the author.

Acknowledgements

I would like to express my special thanks to my supervisor Henry Thompson for his patience, guidance, and for providing me with invaluable feedback throughout the project.

Thanks to everyone who contributes to Common Crawl and makes the existence of this incredible resource possible.

Use of Cloud computing facilities for the work reported here was supported by Microsoft's donation of Azure credits to The Alan Turing Institute. This work was supported by The Alan Turing Institute under EPSRC grant EP/N510129/1.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Lukasz Domanski*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Web and the Internet enable communication, sharing resources, and exchanging knowledge on an unprecedented scale and, for most people in developed countries, the Web has become an essential part of their everyday life [21]. Because of its role in today's society, investigating the Web's properties is a popular area of research; however, due to the Web's heterogeneity and its decentralised nature, such research presents numerous difficulties [21]. The process of obtaining a sample of the Web, called Web crawling, presents significant technical, legal, and ethical challenges [3, 5, 23, 21].

Common Crawl is a publicly-available multi-petabyte dataset which contains billions of webpages [21] and which allows researchers and organisations to "bypass" the challenges of web crawling and simply download the required data free of charge. The data, collected over the past 14 years, brings "a copy of the internet to internet researchers, companies and individuals at no cost for the purpose of research and analysis" [44, 21]. One of the biggest advantages of Common Crawl is its enormous size as it enables research which would otherwise be impossible, such as longitudinal studies which require data collected over many years.

However, processing large datasets often requires using multiple machines and involves a whole array of technical challenges including cluster management and monitoring, scheduling jobs, handling hardware failures, and so on. Many data processing frameworks were created to solve the problem and one such framework, **Apache Hadoop**, became immensely popular and is consider by many to be the "de facto standard in both industry and academia" [18] for efficient batch processing of large scale

data.

The work done during the first part of this project, which focused on whether Common Crawl is a representative sample of the Web as a whole, raised a range of more practical questions regarding the most efficient way to process the dataset. Scalability is seen by many as "the most important feature of a distributed data processing platform" [32] and Hadoop excels in that regard as it can scale effortlessly to thousands of cores. But the question remains: how substantial are the overheads introduced by the framework? Last year, the author used Apache Hadoop to stream and process hundreds of terabytes of data. Was that the right choice?

Analysing Common Crawl involves processing terabytes of data, but often the computation that needs to be performed is very simple - such as counting occurrences of certain words. Those kinds of jobs, which the author dubbed **scan-and-count** workloads, are the focus of this report as they are very common and have many shared characteristics. The main goal of this project was to identify more efficient alternatives to Apache Hadoop and quantify how much faster we can expect them to be for scan-and-count workloads. Moreover, this report goes beyond the software used to process the data and looks at the underlying hardware and its impact on performance.

A benchmark scan-and-count job was selected and implemented using a range of bash utilities in order to establish a good point of reference for comparisons with other solutions and frameworks. The experiments showed that GNU parallel [38] used with `unpigz` [1] utilises the available hardware efficiently and provides a good indication of what is the best possible runtime given the available resources.

The CPU choice can have a large impact on the runtime. Intel Xeon Phi "manycore" processors, which allow running a very high number of parallel threads, were investigated in the hope that a high degree of parallelism might be beneficial for scan-and-count workloads which might seem to be very CPU-light. However, Common Crawl data is stored compressed in `.gz` format and, as decompressing large files is computationally expensive, the tested Intel Xeon Phi processor performed extremely poorly due to low base clock speeds and poor single-core performance.

A common "rule of thumb", which says that the number of concurrent tasks should be set to the number of available logical cores, was empirically verified. The results show that, in most situations, this is a good strategy as it provides optimal performance for CPU-bound tasks and nearly-optimal performance for I/0-bound tasks.

Next, a pseudo-distributed single-node Apache Hadoop setup was benchmarked against a bash tools implementation (or BTI for short) using multiple workloads. BTIs outperformed corresponding Hadoop jobs in all tested scenarios and achieved runtimes from 35% to 70% lower than Hadoop which was shown to use the hardware inefficiently.

Further comparisons were performed between Apache Hadoop, Apache Spark, and BTIs on a 4-node HDInsight cluster hosted on Microsoft Azure. Firstly, as Apache Spark is written in Scala but offers APIs for Scala, Python, and Java, Spark jobs written in those three programming languages were stacked against each other and significant differences in runtimes were observed in some circumstances. The fastest Spark implementations were compared with the corresponding BTIs and Hadoop implementations. Hadoop was found to be slower than Spark and BTIs by a significant margin.

The aforementioned results show that Hadoop is a highly scalable, but not necessarily a highly performant framework. However, because of the immense popularity of the framework many users might choose it because the existing code uses Hadoop, or because they are familiar with the programming model if offers.

This led to the question of whether it is possible to allow using existing Hadoop Streaming codebases while providing superior performance. The author developed Wee Common Crawl Utility (or `wecu` for short) - a bash utility which automates common, tedious tasks related to analysing Common Crawl data, allows running MapReduce scan-and-count jobs without writing any code, and supports existing Hadoop Streaming scripts.

In summary, the hypothesis investigated in this report was that a significant runtime reduction can be achieved by replacing Apache Hadoop with an alternative framework, such as Apache Spark or `wecu`. This hypothesis was confirmed by comparing runtimes achieved by the solutions under the investigation in a variety of settings and scenarios including a single-node local setup, a multi-node cluster hosted in the cloud, processing data available locally, streaming data stored remotely, as well as using specialised hardware offering a high level of parallelism.

1.1 Structure of the report

Chapter 2 clarifies some terminology, introduces the dataset and the Big Data frameworks used throughout this project.

Chapter 3 describes the work performed during the first part of the project.

Chapter 4 provides an overview of the existing literature related to Apache Spark, Apache Hadoop, and the performance of those frameworks in different applications.

Chapter 5 describes the methodology used in the project. Chapter 6 describes the performance comparisons between a number of data processing solutions performed in a variety of scenarios.

Last but not least, Chapter 7 describes the development of Wee Common Crawl utility, a highly efficient data-processing framework created by the author. The chapter outlines the features of the framework as well as some of the challenges faced during the implementation.

Chapter 2

Background

This chapter provides the reader with the necessary background information. Section 2.1 clarifies terminology used in the report. Section 2.2 describes how Common Crawl, the dataset used throughout the project, is organised and what kind of information it contains. Finally, Sections 2.3 and 2.4 discuss the most important components and the internal workings of Apache Spark and Apache Hadoop - the two Big Data frameworks used in this project.

2.1 Terminology

This section clarifies terminology which might be ambiguous or unfamiliar to the reader.

- **Webpage** - in this document, the terms **webpage** or **page** always refer to a particular document available for retrieval from some URL (and not, for example, a set of all documents available under a particular domain).
- **(Physical) processor** - in this document, the terms **processor** or **CPU** always refers to a physical CPU, i.e. a physical device installed on a motherboard.
- **(Physical) core** - in this document, the term **core** always refers to a physical core, i.e. to an individual processor within a physical CPU.
- **Logical core** - some CPUs have a feature which allows each core to run multiple threads simultaneously. In this document, the number of **logical cores** refers to the total number of threads the CPU can run simultaneously. For example, a

quad-core Intel CPU with "Hyper-threading", which can run 2-threads per core, has 8 logical cores.

- **Virtual core (vCPU)** - in the context of virtual machines, vCPU is a virtual representation of a single CPU core. Depending on the hypervisor used, one physical core can correspond to many vCPUs which can all run on the same core.
- **Parallel/simultaneous task** - often the computation can be sped up by performing a number of operations in parallel, at the same time, by running a number of threads or processes simultaneously. For the purposes of this report, there is often no need to make a distinction between a thread and a process and that's why the term **task** is used instead.
- **Amazon Simple Storage Service (Amazon S3)** - a cloud-storage solution which offers impressive scalability, as well as security, data availability, and so on. Common Crawl uses Amazon S3 to store the dataset as part of their **Public Dataset Program**. Thanks to that program, storing the data is free of charge for the Common Crawl organisation and, equally importantly, the dataset can be downloaded for free with no bandwidth charges.

2.2 Common Crawl

Common Crawl is a multi-petabyte dataset which includes billions of webpages collected over the past 12 years. This section briefly describes how the dataset is structured.

The data is collected by crawling the Web monthly, and some crawls contain more than 300TB of data. Each month, a number of pages from a "fetch list" generated in advance are downloaded and saved in 3 data formats: `WAT`, `WET`, and `WARC`. The raw HTTP response is saved to a `WARC` file. The response then undergoes data extraction and: 1) if the document is an HTML webpage with textual content, the detected text is saved to a `WET` file and 2) metadata related to the response (for example, a list of outgoing links) is saved to a `WAT` file. All those files are compressed and stored on AWS S3.

2.3 Apache Hadoop

Apache Hadoop is a collection of utilities, a framework for processing large-scale data [24]. It consists of 3 major components [34]:

- **MapReduce**

  MapReduce is a data-parallel programming model (and the associated implementation) in which data processing is separated into two stages: **map** and **reduce** [39]. In the **map** stage, the input data is read from the disk and gets mapped to a set of intermediate key-value pairs. Those key-value pairs are sorted by the key and saved on the disk, and later, in the **reduce** stage, they are merged to produce the final output [14]. The programmer only needs to write the logic of those 2 stages, and the complexity of dealing with distributed systems is abstracted away by the framework. Parallel execution of the map tasks is equivalent to a serial execution, so the programmer does not need to consider any of the issues related to concurrency or parallel programming. [21, 34]

- **Yet Another Resource Negotiator (YARN)**

  YARN is a cluster management framework which is in charge of scheduling jobs, handling failures, load-balancing, allocation of resources, enforcing execution limits and invariants, and other cluster-management tasks [34].

  YARN uses a master-slave architecture. There are a few important daemons. The master runs a *ResourceManager* (which tracks resource usage, monitors failures, enforces allocation invariants) and *ApplicationManager* (which requests resources from the *ResourceManager*, generates a concrete execution plan given available resources, and monitors its execution). [43]

  Each slave machine runs a *NodeManager* which handles resource allocation on that slave and communicates with the *ResourceManager*. [43]

- **Hadoop Distributed File System (HDFS)**

  HDFS is a distributed file system suitable for storing large-scale data. It is designed with maximising throughput and resilience to failures in mind, so it is suitable for use on commodity hardware. HDFS uses master-slave architecture as well. A single *NameNode* daemon, which manages the filesystem, runs on the master machine, and each slave machine runs a *DataNode* daemon which

handles all data operations on that machine. Data can be replicated on multiple machines so that hardware failures can be handled. [8]

2.3.1 Hadoop Streaming

Apache Hadoop was created using Java and the map and reduce stages would normally need to be implemented in Java as well. **Hadoop Streaming** is a widely used feature of the framework which allows writing MapReduce jobs using any programming language. Typically, a pair of executables is provided: one for the **map** phase and one for the **reduce** phase. The framework uses Linux pipelines to pass data to/from those executables that read the input from `stdin` and emit the output to `stdout`. Executing the Hadoop job is then equivalent to a serial execution of the following command[1]:

```
cat input_files | ./map_executable | sort | ./reduce_executable
```

2.4 Apache Spark

Frameworks designed for batch processing data (such as Hadoop) assume an acyclic data flow and that makes them unsuitable for some workloads, such as iterative algorithms or interactive queries. That's why Zaharia et al. [45] proposed a new framework called **Spark**, which is suitable for those use cases but also retains the guarantees provided by MapReduce such as scalability and fault tolerance.

To achieve that, the framework utilises so called **resilient distributed datasets (RDDs)** - read-only collections of objects partitioned across a set of machines that can be cached in memory for quick access and are created by reading data from files (located on HDFS, local file system, or any other Hadoop-supported file system) and transforming it. Fault tolerance is achieved by storing enough information about how an RDD was derived to be able to rebuild just the partitions that were lost (from the input data stored in persistent storage). [45, 40]

Spark is implemented in Scala, but it provides APIs for Java and Python as well. The Python API is often referred to as **PySpark**.

[1] It should be noted that this is a slight oversimplification for illustrative purposes.

In terms of performance, Spark really shines in applications where a "working set" of data is reused repeatedly as, in contrast to Hadoop, it can store that data in the cluster's memory. However, Spark can also work with datasets larger than the available memory, albeit at reduced performance. [45]

Aforementioned Spark RDDs are a "core but old API" [40] and there are other data structures available which might be more suitable in some use cases. **Spark SQL** is a newer API that, in contrast to RDDs, is more space-efficient and gives the framework more information about the stored data. This allows for additional optimizations that can be performed internally by Spark. **Dataset** is another interface that combines the benefits of the optimised execution engine offered by Spark SQL with the benefits of RDDs (such a strong typing). Last but not least, **DataFrame** API provides a "table-like" abstraction that is similar to Datasets but is organised into named columns and provides convenient function calls like select, filter, groupBy [40, 6]. The consequences of the API choice go beyond user preference and convenience as Karau [27] and Behera [4] showed that DataFrames can be multiple times faster than RDDs when using PySpark.

2.4.1 Execution planning

Apache Hadoop requires that, if the workload cannot be expressed as a single MapReduce job, the user divides the workload into a "pipeline" of Hadoop jobs executed serially, one-after another, in which the later jobs operate on the output from the previous job(s). Spark takes a more "declarative" approach and the framework can determine which tasks can be executed in parallel and what the possible optimisations are. [41]

Firstly, when the user submits the job to Spark, the data transformations in the workload are identified and arranged in a **Direct Acyclic Graph (DAG)** according to the flow of the program. The DAG acts as a **logical plan** and is used to construct a **Physical Execution Plan** which specifies how the computation will be performed in more detail. [41]

2.4.2 Spark and Python

It is essential that the reader is aware of the basic architecture of Apache Spark as this report discusses performance issues related to usage of different programming

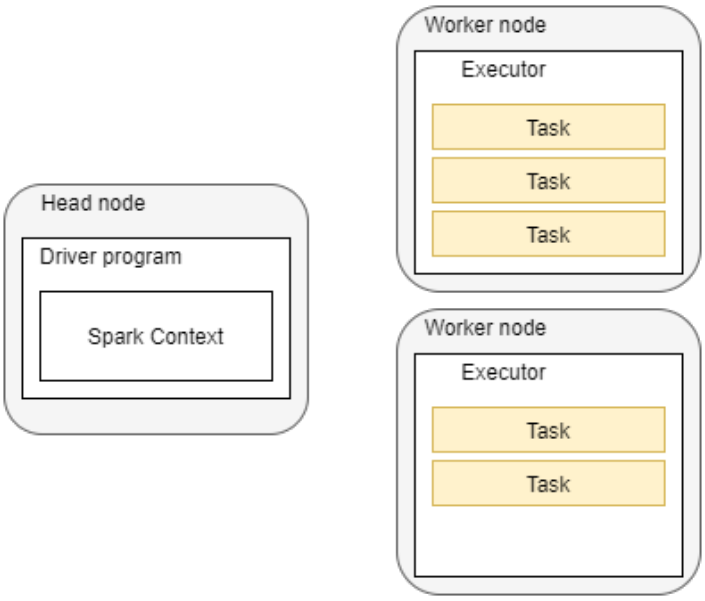

Figure 2.1: Apache Spark execution model when Scala or Java are used. The figure was created based on information obtained from [28] and [10].

languages with the framework. Both Scala and Java are compiled to **Java Byte Code** which is executed by the **Java Virtual Machine (JVM)**. The execution model used for jobs written using Scala or Java is (almost) the same and is shown in Figure 2.1.

Firstly, a `SparkContext` object, a Scala implementation entry point of a Spark application (or the Java wrapper - `JavaSparkContext`), needs to be created within the driver program running on the head note. `SparkContext` is then used to describe operations on input data. A program describing those operations is then translated into a DAG and then further into stages (or a physical execution plan). The work is then delegated to processes running on the worker machines. Each worker machine runs a number of **Executors** (which are JVM processes) and each Executor can run a number of tasks in parallel. The code, both on the head node and on the worker nodes, is running entirely within the Java Virtual Machine. [28]

When Python is used to write the code, the execution model (shown in Figure 2.2) becomes somewhat more complicated. A `SparkContext` is created within the Python code and is used to define a set of transformations that will be applied to the data. However, those operation are not performed directly in Python and they are “translated” into operations on the `SparkContext` existing in the JVM. A library called `Py4J` is used to operate on Java objects from the Python code - the Python objects themselves are

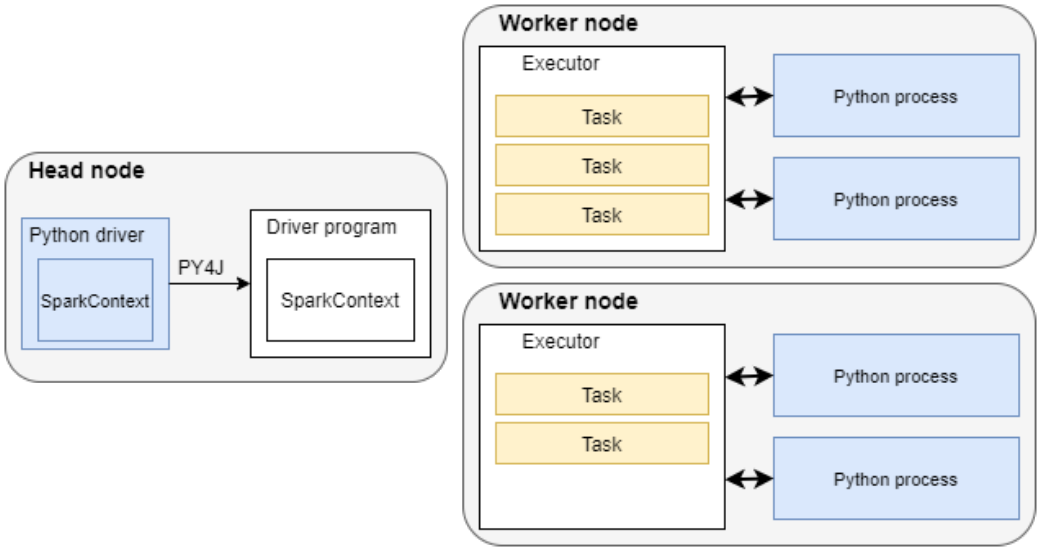

Figure 2.2: Apache Spark execution model when Python is used. The figure was created based on information obtained from [28] and [10].

merely thin wrappers around the corresponding Java objects. This means that, as long as the data transformations in our code are limited to the operations provided by the Spark API, the computation will be performed entirely within the JVMs. [28, 10]

However, if that is not the case, additional Python processes spawned on the worker nodes are used to run Python code which cannot be "translated" to Java. The communication is established between the JVM executor thread and the Python process using sockets. The required data is passed between the processes by serialisation and deserialisation. The Python functions which perform the computation are also transferred to the worker machines by serialising the entire function closure. [28, 10]

As a result, in comparison to the operations performed using `Py4J` and Spark API, using the Python processes to run computation is more expensive because of the overheads related to communication and data serialisation. [10]

Chapter 3

Previous Work

The overarching goal behind the work done in both parts of the project was to provide Common Crawl users with the knowledge required to effectively work with the dataset. The work done this year focused on the technical aspects of processing large scale data. However, last year, the focus was on how useful the results obtained using the dataset are. In particular, the author attempted to establish whether Common Crawl is a representative sample of the Web as a whole.

Many authors examined Common Crawl's representativeness before, however, their analyses were restricted to a small subset of the Web, or even a single statistic. The last year's project aimed to address the shortcomings of the existing literature by performing a more general analysis. This was done using a methodology based on comparisons between different datasets.

A comparative analysis of multiple large-scale datasets was performed. The analysis, which included a wide range of statistics, unveiled a number of inconsistencies between Common Crawl and other datasets. Further analysis revealed that most of the statistics under investigation were affected by the crawling process which means that, for those statistics, we cannot make inferences about the Web as a whole based on Common Crawl (or any other web crawl).

In addition to the comparative analysis, last year's report explored related important theoretical topics such as the impact of dynamically generated content and spam pages on the representativeness of Common Crawl and web crawls in general. The author empirically verified that those issues have a non-negligible impact on the dataset and need to be taken into consideration when conducting research.

This report looks into the technical aspects of efficient data analysis like the aforementioned cross-dataset comparisons, which required processing terabytes of Common Crawl data.

Chapter 4

Related Work

Maximising throughput is often stated as one of the primary design goals of Apache Hadoop [36] so it is unsurprising that many authors work on the assumption that, when it comes to batch processing data, Hadoop is the most efficient solution and that it will outperform other frameworks [25]. However, careful analysis of the existing literature reveals that this is not the case, and that Hadoop should be seen as "a highly-scalable, but not necessarily a high-performance cluster solution" [26].

Pan [34] carried out a performance comparison between Apache Spark and Apache Hadoop using a cluster of Virtual Machines. The author compared both frameworks using a benchmark job which counted the number of occurrences of each word in the dataset (`WordCount`) and another job which additionally sorted the output by the number of occurrences of each word first, and then by the word itself (`SortedWordCount`). The author showed that Spark outperformed Hadoop by at least a factor of 3 for the `WordCount` job and was at least 6 times faster than Hadoop for the `SortedWordCount` task. This study, however, uses a very small 2GB dataset which exacerbates the impact of start-up overhead, and hence the findings might not apply directly to processing multi-terabyte datasets.

Jiang et al. [31] selected a variety of Big Data workloads of different kinds and studied micro-architectural and architectural behaviours of Apache Hadoop and Apache Spark when running those workloads on a 17-node Xeon cluster. The authors found that Hadoop runtimes were from 2.7 to 8.4 times longer compared to Spark. Spark was found to have a higher average disk utilisation and better spatial and temporal localities of disk accesses. The comparison included a **BigDataBench Grep** workload that has

similar characteristics to scan-and-count jobs. Interestingly, Spark was over 3 times faster than Hadoop for the BigDataBench Grep job. Last but not least, the authors conclude that "current Intel commodity processors are sufficiently efficient for in-memory computing" [31].

McSherry et al. [32] considered the hidden cost of parallelisation. In their paper, the authors proposed a new metric for Big Data platforms: **Configuration that Outperforms a Single Thread** (or **COST**). This measure, they argue, weights the gain in scalability against the resulting overheads. The authors discovered that many data-parallel systems have "either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations" [32]. The authors remark that few papers "directly evaluate [the] absolute performance [of a system] against reasonable benchmarks" [32].

Drake [22] attempted such a comparison and demonstrated that a single threaded implementation based on bash tools can outperform an equivalent **Hadoop** implementation by a factor of 47, and a multi-threaded bash implementation can outperform the framework by a factor of 234. It should be noted, however, that Drake's work is limited to one particular task, uses a small dataset (less than 2GB), and lacks the details required to replicate those findings and to explain **Hadoop**'s suprisingly poor performance.

Ding et al. [17] investigated the performance of **Hadoop Streaming**. The paper outlines how using pipes to pass data around introduces significant overheads: transporting key-value pairs between Hadoop (a Java process) and the executable used to process the data requires two copy operations which can only be performed sequentially: "from the Hadoop's memory area to pipe buffer and from pipe buffer to the external executables memory area" [17]. The authors quantitatively assessed those overheads - a data-intensive Hadoop job counting the number of occurrences of each word in a 20GB dataset took over 26 minutes using Hadoop streaming and less than 15 minutes using the Java API (over 75% runtime increase). However, the authors showed that for very computationally intensive jobs, streaming can actually be faster than using Hadoop's Java API because other programming languages might be more efficient in those jobs [17].

Apache Spark is becoming more and more popular and hence it is a topic of many publications as well. Spark is implemented in Scala but it provides APIs for Java,

Scala, and Python. It is interesting to consider whether the choice of a programming language will have a non-negligible impact on performance of the framework. To the best of the author's knowledge, a direct, quantitative comparison of those languages when used with Apache Spark has not yet been performed. However, Bhat et al. [7] studied "the time taken by the same data structure in various technologies", including Python and Scala, and the authors found non-insignificant differences for some use cases.

Dimakopoulos [16] performed an impressive large-scale empirical evaluation of Spark's scalability. The framework was evaluated on inputs ranging from 20TB to 1PT and using between 150 and 4000 virtual cores. The workload used in the experiment was a network-bound, memory-hungry "physics processing example". This makes Dimakopoulos' work very relevant to the topic of this report, as scan-and-count workloads can often be network- (or disk-) bound as well. The author observed efficient use of hardware for all input sizes and a linear behaviour between input size and runtime. Furthermore, a convergence was observed when scaling beyond some number of cores or memory - further increase in the resources available brought diminishing returns.

Moreover, Shahrivari [36] showed that Spark outperforms Hadoop in a range of different use-cases (including word counting and a "grep" workload) and Hazarika et al. [25] showed that Spark can outperform Hadoop in a "pseudo-distributed" single-node set-up as well.

In conclusion, it is clear that, in terms of performance, Hadoop is not always the best available solution. This report investigates this issue further in the context of scan-and-count workloads and explores alternatives which do not require rewriting the existing codebase or learning a new framework because, as explained in further chapters, switching frameworks can often present difficulties. Last but not least, the existing literature does not address the performance differences between the Spark APIs in different programming languages adequately. This report addresses this shortcoming by performing a comparison of all available APIs.

Chapter 5

Methodology

This chapter describes the methodology used in this project by detailing the benchmark workload used for the comparisons, the available computing resources, and the procedures used to measure hardware utilisation and runtimes.

5.1 Benchmark workload

To quantitatively compare performance of different data-processing solutions, we can compare runtimes of a sample "benchmark" task. This task needs to be chosen carefully so that it represents the typical needs of a Common Crawl user because tools are typically designed with specific kinds of workloads in mind. When analysing `WARC` files, we are often interested in computing some kind of a statistic such as a number of occurrences of certain words, internet protocols, file extensions, and so on. Those workloads, which scan the entire dataset and perform a simple computation and which the author dubbed "**scan-and-count workloads**", share a few characteristic traits:

1. The required computation is very simple so we expect those workloads to be network-bound or disk-bound, however, depending on the tools used, this might not be the case (see Section 6.1.1).
2. The job is data parallel which means that the tasks can be easily and evenly distributed among the processors by splitting the input data.
3. The computation is not iterative.
4. The output size is typically small compared to the input data.

Machine	No. of vCPUs	Memory	Storage	Price (as of January 2019)
Worker	8	56 GiB	400 GiB	£0.4457/hour
Head	4	28 GiB	200 GiB	£0.2229/hour

Table 5.1: The parameters of the HDInsight virtual machines used.

Other common types of workloads include CPU-intensive tasks such as analysing crawled PDF files or iterative algorithms such as Page Rank; however, those kinds of workloads are not the focus of this report due to time constraints.

The author decided to use a job that counts the number of crawled webpages included in each `WARC` file because it can be implemented as counting the number of times the `^WARC-Type: request` regular expression matches each file, and hence it embodies all of the characteristics mentioned above. Additionally, this makes the benchmark easy to implement and, because this project involves comparisons between a number of different tools and programming languages, choosing a simple benchmark job decreased the chances of an implementation error affecting the results.

A decision was made that the benchmark job should aggregate the result (i.e. sum the results from all of the files). Some of the tested tools do not offer an easy way to display the output per file, and thus the author decided to aggregate the results in all jobs for consistency.

5.2 Available resources

This section describes the computational resources used to run the experiments.

5.2.1 HDInsight

HDInsight is a cloud-based data processing service offered by Microsoft Azure. HDInsight allows quick creation of clusters which come with commonly used software, such as Apache Hadoop, Spark, and Kafka, pre-intalled and pre-configured. This removes the need for complicated cluster set-up and performance tuning [20, 21]. A cluster with 2 **head nodes** (which coordinate the operation of the cluster) and 4 **worker nodes** (which run the computation) was used, with the parameters shown in the Table 5.1.

Azure servers use a 12-core Intel Xeon E5-2673 v3 2.4 GHz (Haswell) or a 20-core E5-2673 v4 2.3 GHz (Broadwell) processors that can can go up to 3.1 GHz using the Intel Turbo Boost [19]. The four worker machines have 8 vCPUs each so, in total, 32 tasks can run in parallel.

The cluster did not have access to locally stored Common Crawl data and hence streaming was required.

5.2.2 elli

An Informatics-owned machine called `elli` was used extensively throughout the project. The machine is equipped with over 153 Terabytes of disk space and two Intel Xeon CPU E5-2620 v4 2.10GHz processors that can go up to 3.0GHz using Intel Turbo Boost. Each processor has 16 logical cores so `elli` can run 32 threads in parallel.

This machine has direct access to a RAID0 disk array containing multiple months of Common Crawl data, including the August 2019 crawl which was used in all experiments.

5.2.3 mill8

Another Informatics-owned machine called `mill8` was used as well. This machine does not have `elli`'s impressive storage capacity but it features 110 gigabytes of RAM and Intel Xeon Phi CPU 7285 @ 1.30GHz processor. The CPU is a so called "many-core" processor and has a relatively low clock-speed (up to 1.40GHz with Intel Turbo Boost) but it has 68 cores, each one capable of running 4 threads in parallel - so it is able to run up to 272 parallel threads.

`mill8` has access to the drives containing Common Crawl data (which are connected to `elli`) over the local network. The author suspected this might impact the runtime; however, as explained in Section 6.1.3, the Xeon Phi CPU turned out to be a significant bottleneck and further investigation into the local network bandwidth was not necessary.

5.3 Measurements

This section details the methodology used to measure hardware utilisation and runtimes of the tested workloads.

5.3.1 Measuring hardware utilisation

Hardware utilisation can tell us if a data-processing solution operates at its full potential or whether it was misconfigured and is using the hardware inefficiently. It can also tell us if there is any potential for improvement - for example, no performance gains are possible if a disk-bound job already achieves 100% disk utilisation.

The CPU utilisation was monitored using the `iostat` bash utility. Initially, `iostat` was used to monitor disk usage as well, however, the results were often unexpected and contradictory. Upon further investigation, it became clear that the utilisation returned by `iostat -x` (`%util` field) is essentially meaningless if a RAID disk array is used [9]. Therefore, disk throughput was measured instead of disk utilisation. In this report, **all of the graphs showing disk usage use the same scale** to allow for quick visual comparisons.

Hardware utilisation was measured every second over 10 minutes while each implementation was running. Measurements were done 5 minutes after starting the command to avoid capturing any anomalies related to start-up overheads or the final moments of the job when only a few tasks are running. Disk utilisation was monitored for a fixed amount of time in order to make the results easier to compare directly between workloads with drastically different runtimes. As even the most inefficient solutions described in this report process at least 5 files per minute, monitoring the utilisation for 10 minutes is more than sufficient to spot any patterns that might emerge, such as drops in throughput due to straggler tasks, etc.

5.3.2 Measuring runtime

To make the runtimes comparable between different tools, as many factors as possible were kept constant between the experiments:

- The runtimes were measured using `time` bash utility which measures the time

between running a command and when it returns. This means that the runtime includes start-up overheads. For example, for Apache Hadoop the runtime includes job scheduling and resource allocation. This can be seen as "a bug or a feature", but in any case this should have a negligible impact on the results as the input data is large.

- A sample of the first 1000 WARC files from the July 2018 crawl was used as the input data. On average, a single `WARC` file takes approximately 0.85GB when compressed (or 4GB uncompressed) so the dataset used for experimentation consist of around 850GB of compressed data (or 4TB after decompressing). The files stored on `elli` were used for experiments involving local data.
- The author did not have exclusive access to the Informatics-owned machines. To minimise the impact of other workloads, the jobs were scheduled when the machines were least utilised. Each job was re-run 3 times and the runtimes were averaged.

`WARC` files contain one line equal to "`WARC-Type: response`" for each crawled page. In this project, this was exploited to count the crawled pages by counting the lines starting with that string. Of course, lines *equal* to "`WARC-Type: response`" (and not *starting with* that string) could be counted instead, however, considering that the number of lines containing that string are small relative to the total number of lines in each file, the runtime differences between the two should be negligible and the author (arbitrarily) decided to use the latter approach. As some crawled pages may contain the string "`WARC-Type: response`" in their content, the results produced with this simple method might be slightly inaccurate, however, this is not a problem since only the runtimes are relevant to the subject of this report and the performance impact of this issue is negligible.

Chapter 6

Processing large-scale data

This chapter describes a number of runtime comparisons performed to establish what are the most efficient ways of processing large-scale data. The chapter starts with Section 6.1, which describes the experiments performed using a single machine. In Section 6.2, a cluster with four worker machines is used to verify that the conclusions drawn from the results obtained using a single machine hold for multi-node clusters as well.

6.1 Data processing using a single machine

This section details the experiments which used a single machine in order to investigate a range of issues related to processing large-scale data in detail. Section 6.1.1 investigates a range of bash utilities which can be used to process the dataset and demonstrates that the best performing set of tools uses the hardware efficiently. Section 6.1.2 investigated the impact of data locality by comparing processing local data with streaming files. Finally, Section 6.1.3 investigates the suitability of Intel Xeon Phi processors for scan-and-count workloads and Section 6.1.4 discusses the optimal number of concurrent tasks that should be used with scan-and-count workloads.

Last but not least, Apache Hadoop is compared with an implementation using bash utilities in Section 6.1.5 in order to investigate the overheads introduced by the framework.

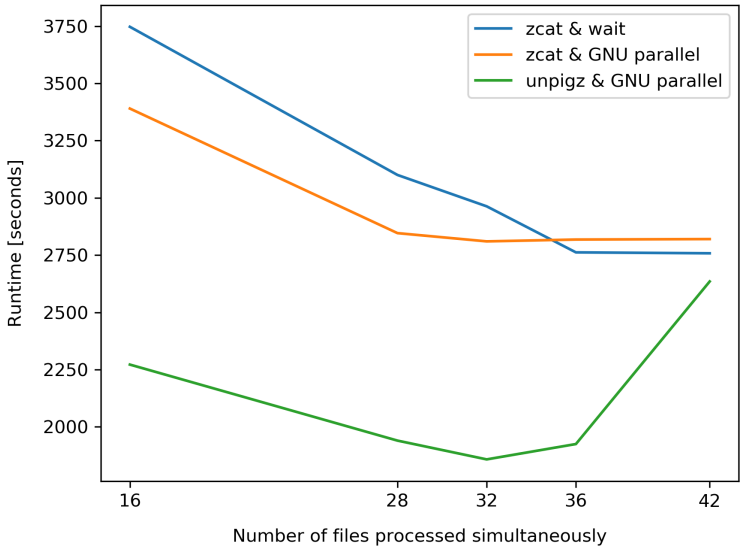

Figure 6.1: Performance comparison of different bash tools. One thousand locally stored WARC files (approximately 850GB of compressed data) were processed using each set of tools. Comparison was repeated for different numbers of files processed simultaneously.

6.1.1 Bash tools

The benchmark workload was implemented using a variety of bash tools to establish a good point of reference for comparisons with other solutions. Linux piping was used to stream the data between `unpigz` and `grep -c` which counted the number of times the `^WARC-Type: response` regular expression matched each file (WARC files include one such line per crawled page).

Testing began with a naive implementation (1) that uses `zcat` to decompress the files and a combination of bash `&` operator and the `wait` command to parallelise the execution. This solution is simple but it is susceptible to the straggler task problem - some tasks take much longer than others and stop the computation from advancing. Implementation (2) used `zcat` as well, but GNU parallel, a powerful command line utility that allows running commands in parallel using a single or multiple machines, was used instead of `&` and `wait`. The third implementation (3) used GNU parallel, however, `unpigz`, a lesser known but more efficient alternative to `zcat`, was used.

The implementations were tested on 1000 `WARC` files. The experiments were performed on `elli` (see Section 5.2.2). Each set of tools was tested with 16, 28, 32, 36, 42 parallel

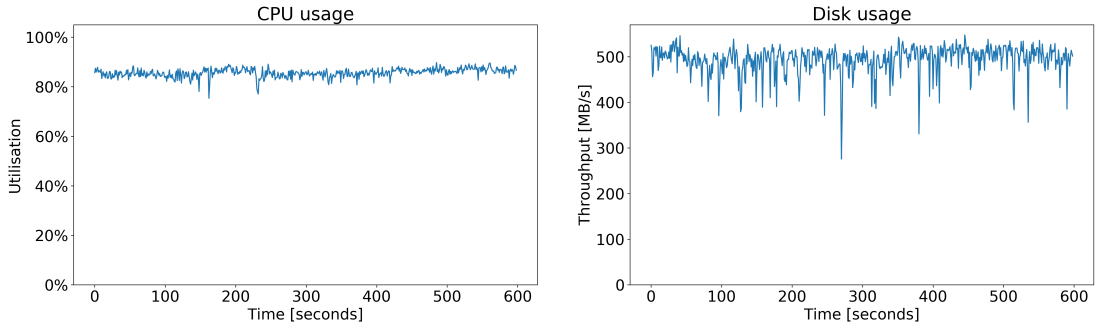

Figure 6.2: Hardware utilisation measured while a bash-tools-based implementation using GNU parallel and unpigz was running

tasks (note that `elli`'s CPU has 32 logical cores).

The results are visualised in Figure 6.1. Y-axis shows the runtime (in seconds) and the x-axis shows the number of parallel tasks. Implementation (3) using `unpigz` and GNU parallel is a clear winner. On average, when using 32 threads, it is capable of processing almost 4TB of (uncompressed) data in less than 31 minutes. We can clearly see that, beyond a certain point, the runtime plateaus and increasing the number of simultaneous tasks does not increase the performance (or even degrades it). Based on those results, the author decided to use GNU parallel exclusively for the reminder of this project. This is because 1) it is faster than (or nearly as fast as) other implementations for all tested scenarios, 2) it makes writing code easier, and 3) it is capable of running the computation on multiple machines.

Hardware utilisation was measured for implementation (3) using 32 parallel threads and the results are shown in Figure 6.2. The x-axis is the CPU utilisation (graph on the left) and the disk usage in kilobytes per second (graph on the right). The implementation is disk-bound but CPU usage is high as well. It is clear that no further runtime improvements are possible as implementation (3) fully utilises the capabilities of the hardware, and its runtime can be considered a "lower bound" for what is possible to be achieved using `elli`.

6.1.2 Data locality

In the previous section, we established that `unpigz` and GNU parallel can fully utilise the available hardware and process locally stored data very efficiently. However, due to its size, the required data can rarely be stored locally. Common Crawl is hosted on

Simultaneous tasks	Local data	Streaming	Ratio
16	37m52s	244m59s	6.4
32	30m58s	177m1s	5.7
42	43m55s	172m42s	4.0

Table 6.1: Runtime comparison between processing 1000 WARC files stored locally and streaming them from Amazon S3. The last column was computed by dividing the time it took to stream and process the data (column 3) by the time it took to process the local data (column 2).

AWS S3 which allows streaming data over HTTPS using tools such as `curl` while it is being processed. The answer to the question of how much slower it is to process the dataset like that (compared to processing data stored locally) depends heavily on the available internet connection.

The experiments performed on `elli`, which utilised the benchmark workload (3) re-implemented using `curl` on 1000 WARC files, show that, depending on the number of concurrent tasks, streaming data is 4.0-6.4 times slower (Table 6.1). However, a similar (but smaller due to storage limitations) experiment using 10 WARC files performed on an Azure virtual machine (and repeated 5 times) showed a difference in runtime **smaller than 3%**.

This difference could be due to the geographical location (both data centres hosting the virtual machine and Common Crawl data are located in the United States), the performance of the infrastructure connecting the data centers, and many other factors. In fact, it is difficult to say exactly how fast the internet connection available to the VM running on Azure is. The network is shared between many virtual machines and users, and there are no direct limits on the ingress network traffic on Azure [2]. Therefore, it is possible that the network speed might vary significantly based on current load.

The results presented here make it clear that no general conclusions can made, and it is up to the user to benchmark the available internet connection and act accordingly.

Simultaneous tasks	Local data	Streaming
32	148m45s	181m30s
50	147m42s	175m20s
252	158m50s	161m38s

Table 6.2: Runtime comparison between using `mill8` to process 1000 WARC files stored on `elli` and streaming them from Amazon S3

6.1.3 "Bringing out the big guns" - Intel Xeon Phi

As described in Section 5.2.3, `mill8` features Intel Xeon Phi "manycore" CPU. Those specialised processors are optimised to provide a high-degree of parallelism at an expense of per-thread performance. `mill8`'s CPU has 68 physical cores which can run 272 threads in parallel. This section explores whether Intel Xeon Phi processors are suitable for **scan-and-count** workloads. Considering `elli`'s hardware utilisation, described in the previous section, it seemed unlikely because the workload (3) is disk-bound and hence adding more parallel tasks is not going to help. However, it was not so clear how `mill8`'s performance is going to be for streaming files.

The same benchmark workloads as those described in Sections 6.1.1 and 6.1.2 were used. Table 6.2 outlines the results. Xeon Phi processor turns out to be a major bottleneck when processing local data, slowing down the computation done with 32 simultaneous tasks by a factor of 5. In fact, the bottleneck is so extreme that there is surprisingly little difference in terms of runtime between streaming and processing local data. Those results are not surprising considering that decompressing WARC files is a computationally expensive task and that Xeon Phi's have a very poor per-core performance. Interestingly, streaming data benefits (to a small extent) from the additional parallel tasks. Using 272 parallel tasks, the job takes approximately 10 minutes less than on `elli` using 42 threads.

6.1.4 Choosing the right number of threads

When it comes to choosing an appropriate level of parallelism for the application, a commonly mentioned advice is to set the number of parallel threads to the number of available logical cores [30, 29].

The results presented in the Section 6.1.1 show that this is a good heuristic as, for CPU-bound workloads and when using a commodity CPU, it results in optimal performance. In other scenarios this might not be the case - for example, when processing local data on `mill8`. However, for all of the tested workloads, the resulting runtimes were less than 10% (and in most cases, less then 5%) slower the the best runtime recorded. This shows that, even though not universally true, this common advice is sound and, even if it does not result in the best possible runtime, it certainly is a good starting point for further performance tuning.

6.1.5 Single node Apache Hadoop setup

Apache Hadoop comes with a very large set of configuration parameters. Changing those parameters can have a large impact on performance and it requires knowledge of the internal workings of the framework [26]. Hadoop can be configured to run on just a single machine or on a cluster, which be easily scaled to thousands of machines. The framework handles cluster management, scheduling jobs, and provides guarantees such as being highly resilient to failures so that inexpensive commodity hardware can be used. Those features provide little value unless a multi-machine cluster is used but a single-node setup is not uncommon as it is widely used for debugging or development, because the existing code uses Hadoop, etc.

This report aims to answer the following questions:

1. How significant are the overheads introduced by Hadoop? Are they large enough to warrant a rewrite of the existing code? Does it make sense to use Hadoop with a single node outside of testing and development?

2. Hadoop jobs can be implemented natively in Java or, when using Hadoop Streaming, in any programming language. Users often prefer high-level languages such as Python and hence Hadoop Streaming is a common choice, but what price are we paying for that convenience?

3. And last but not least, can we have the cake but also eat it? Can we develop a more efficient solution that still allows us to use an arbitrary programming language?

This section considers questions 1 and 2 using a single-node set-up which is very suitable for investigating those issues as it provides a lot of control and ease of monitoring

hardware resources. Sections 6.2 and 7 address all 3 questions in the context of a multi-node cluster running on Microsoft Azure.

There are two ways to run **Hadoop** on a single node: **a non-distributed mode** in which Hadoop runs as a single Java process or in a **pseudo-distributed mode** where each daemon (as listed in Section 2.3) runs in a separate Java process [39]. **Pseudo-distributed mode** was used throughout this project.

With a single-node setup, the overheads introduced by the framework can be more significant than usual because that single machine will, in addition to performing the computation itself, run all of the Hadoop deamons which communicate through network requests.

Three Hadoop workloads were tested:

1. A Hadoop job implemented natively in Java that used `.startsWith()` to count crawled pages.
2. A Hadoop job implemented using Hadoop Streaming and Python that used `.startswith()` to count crawled pages.
3. A Hadoop job implemented using Hadoop Streaming and Python that used regular expressions to count crawled pages.

Then, the overheads were evaluated by comparing the runtimes of those jobs with three equivalent jobs implemented using bash tools:

1. A bash-tool-based job using grep to count crawled pages.
2. A bash-tool-based job using Python and `.startsWith()` to count crawled pages.
3. A bash-tool-based job using Python and regular expressions to count crawled pages.

6.1.5.1 Hadoop configuration

The first step was to install and configure **Apache Hadoop** on `elli`. Hardware setup and BIOS settings might affect performance [26], however, they were not considered due to lack of access. The configuration was chosen by following the guidelines outlined by Heger [26]. Hardware usage was monitored to ensure no swapping was taking place. Data compression of intermediate/output data was disabled as the CPU

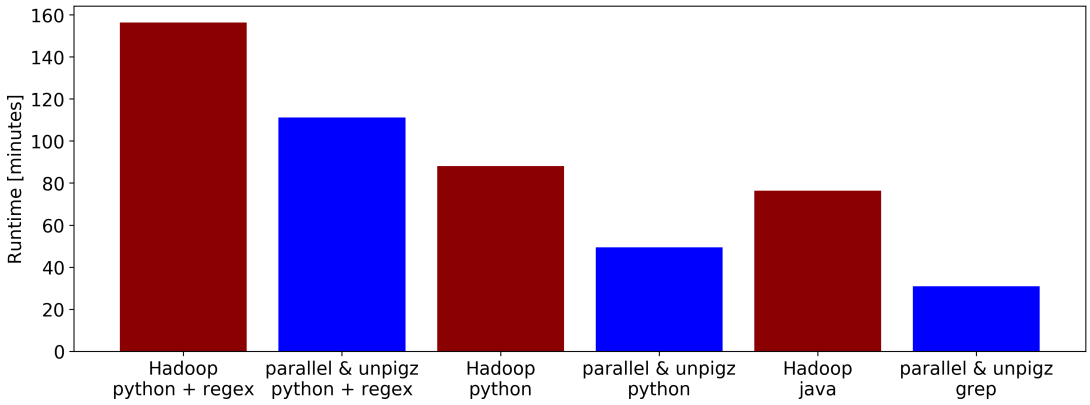

Figure 6.3: Runtime comparison between different variants of the benchmark workload implemented using bash tools and Hadoop.

usage was very high, and jobs which are CPU-bound are likely not to benefit from that feature [26]. The experiments were performed on `elli` using locally stored WARC files. HDFS provides data replication, maximises locality, and maximises throughput but, since a single machine and a RAID disk array were used, those features were deemed not critical. Hence, local filesystem was used instead of HDFS. Using the local filesystem allowed shutting down *NameNode* and *DataNode* daemons and (ever so slightly) decreasing the CPU load. The values for the most important parameters of the final configuration are presented in Table A.1. Two values of the `mapreduce.tasktracker.map.tasks.maximum` parameter that limits the number of parallel map tasks were tested: 32 and 64. Performance was significantly higher with the limit set to 32 so that value was used for the remaining experiments. The bash workloads used 32 parallel tasks as well. The runtimes of the six workloads described above were measured, and the results are shown in Figure 6.3.

Using `unpigz`, `parallel`, and `grep` is the fastest way to process the dataset and the "equivalent" Hadoop job written in Java is 2.7 times slower. Hadoop is also 70% slower than bash tools when using Python and 35% slower when using Python and regular expressions. It is clear that the longer it takes to process a single file (or, in other words, the more computationally expensive the map tasks are) the less significant the overheads introduced by Hadoop become relative to the overall execution time.

Utilisation was measured for the fastest implementation (GNU parallel + `unpigz` + `grep`), the implementation using python and regex, and the Hadoop job that used python and regex as well. The results are shown in Figure 6.4. The x-axis on those graphs shows the CPU utilisation (graphs on the left) and disk usage in megabytes per

second (graphs on the right).

The implementation using `grep` is disk-bound, however, the CPU utilisation is also high. As previously mentioned, this workload uses the available resources effectively and its runtime can be considered a "lower-bound" for what is possible using that particular hardware. Interestingly, the modified version of that implementation, which uses python and regex, is significantly more computationally expensive which makes it CPU-bound and results in significantly lower disk usage. The CPU is an even bigger bottleneck for the Hadoop job.

A major shortcoming of the comparison described in this section is that it involves a single-node setup and hence the reader might wonder whether the conclusions presented here would also apply to clusters with multiple machines. A pseudo-distributed mode was used in which all of the Hadoop deamons normally running on a cluster are executed in separate processes - this simulates distributed operation very closely and, in fact, the results obtained from a multi-node cluster, described in the following sections, confirm the general trends described here.

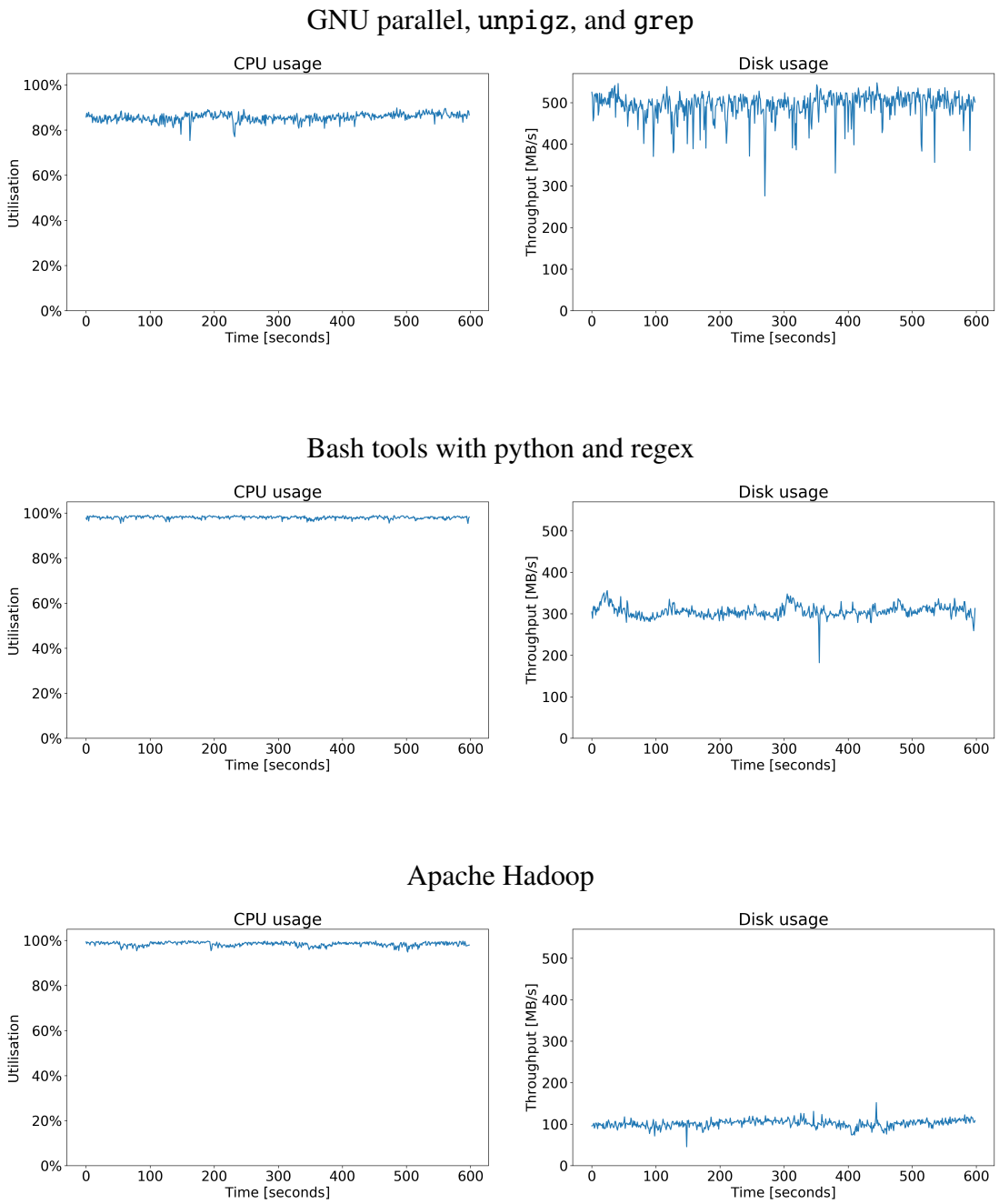

Figure 6.4: Hardware utilisation comparison between a two bash tools based solutions and Apache Hadoop

6.2 Distributed data processing

The previous sections discuss analysing data using a single machine, however, processing large datasets is typically done using clusters with many nodes. Additional cores, memory, as well as disk space and bandwidth are obvious advantages of adding more machines, but the type of the workload influences how easy it is to distribute the computation and how beneficial the additional hardware is going be. Fortunately, **scan-and-count** workloads are trivial to distribute among multiple nodes. The following sections detail the experiments performed using a cluster with four worker machines to verify that the conclusions drawn from running the experiments on a single machine hold for multi-node clusters as well.

6.2.0.1 Bash-tools

The benchmark workload was modified to utilise four worker nodes. GNU parallel requires that the communication between the head node (which is running GNU parallel) and the worker nodes (which run the computation) does not require a password. Passwordless communication was configured and the required dependencies (such as `unpigz` and `parallel`) were installed. The number of simultaneous tasks was manually set to the number of available logical cores (i.e. 8 per worker machine or 32 in total). Lastly, a file with the URLs of a sample of 1000 WARC files was created as well which was then fed into GNU parallel which distributed the computation among the worker machines.

Testing was also performed with 64 parallel tasks (i.e. 16 tasks per worker or 2 per logical core). A delay of 0.1 seconds had to be introduced between starting new ssh sessions in order to avoid overwhelming the worker machines with too many parallel ssh sessions, but this delay should have negligible impact on the runtime. The resulting performance was very poor and hence 32 threads were used for the remaining experiments.

6.2.0.2 Apache Hadoop

Apache Hadoop comes pre-installed and pre-configured on Azure HDInsight clusters created using the `--type hadoop` flag. The default configuration of the clus-

	Java	Spark	Python
Runtime RDD	38m12s	37m40s	143m39s
Runtime DataFrame	29m17s	28m58s	28m36s

Table 6.3: Runtime comparison between different languages which can be used with Apache Spark as well as between using RDDs and DataFrames

ter provides satisfying performance [21] and was not modified beyond changing the `mapreduce.tasktracker.map.tasks.maximum` parameter in order to limit the number of simultaneous map tasks to 32. This was done because the default configuration resulted in over 60 parallel tasks and suboptimal performance.

6.2.0.3 Apache Spark

Apache Hadoop comes pre-installed and pre-configured on Azure HDInsight clusters created using the `--type spark` flag. The default configuration of the framework was used with no modifications, beyond increasing the number of available executors and available cores per executor to allow 32 parallel tasks.

Examining Spark in this context is especially interesting as scan-and-count jobs do not utilise a repeatedly used working set and because the dataset under the investigation is orders of magnitude larger than the available memory on the cluster.

6.2.1 Results and discussion

Two things need to be considered regarding the implementation of a Spark job: the choice of a programming language and an appropriate data type (RDD, Dataframe, or Dataset). The consequences of the data type choice go beyond user preference and convenience as in some cases DataFrames can be multiple times faster than RDDs [27]. Apache Spark is implemented using Scala, but it offers APIs for three languages: Java, Python, and Scala. It is not clear whether there are significant performance differences between the APIs in different programming languages, so a comparison was performed. The benchmark job was implemented using the three programming languages and using both RDDs and DataFrames. The runtimes (using 1000 WARC files) are shown in Table 6.3.

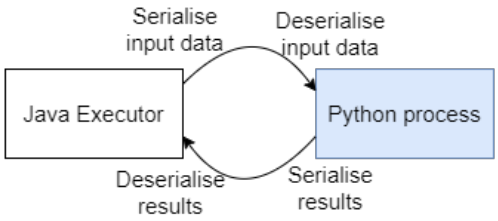

Figure 6.5: PySpark uses inefficient two-way communication which involves serialising and deserialising the data. This is likely to be one of the contributing factors to poor performance of PySpark when RDDs are used.

When Resilient Distributed Datasets (RDDs) are used, both Java and Scala jobs take approximately 38 minutes, however, the Python workload takes *almost 4 times* longer. Moreover, when DataFrames are used, jobs in all three programming languages take approximately 28-29 minutes.

Firstly, based on those results, we can confirm the conclusions drawn by Behera [4] and Karau [27] - using the DataFrames API instead of RDDs led to a significant runtime improvement. But an interesting question remains: why is the PySpark implementation so much slower when using RDDs? The author believes that this can be explained with how PySpark's execution model differs from the execution model of Scala/Java jobs (as detailed in Section 2.4.2). The counting of crawled pages was implemented using `RDDs` as follows:

```
commonCrawlData
    .filter(lambda v: v.startswith('WARC-Type: response'))
    .count()
```

This code includes a lambda function (shown in bold) which has to be executed using a Python process on the worker machine and, as PySpark uses (comparatively) inefficient two-way communication (Figure 6.5), this is likely to negatively impact performance. Moreover, Python executors run in their own processes which could result in higher memory usage due to memory footprint of the interpreter and the loaded libraries.

```
commonCrawlData
    .filter(data.value.startswith('WARC-Type: response'))
    .count()
```

This code uses DataFrame API exclusively and hence the code can be "translated" to

run fully within the Java Virtual Machine. Therefore, inter-process communication is not required and the associated performance overheads are not present.

In summary, the DataFrame API and the powerful optimisation engine available in Apache Spark allow writing code in a more succinct, declarative style. Moreover, the availability of APIs for multiple programming languages give the users more flexibility, and as long as the Python code is limited to built-in Spark API functions, all three languages offer similar performance. However, this flexibility comes at a price, as the programmer should be aware of framework's execution model and care needs to be taken as fairly small changes in the code can have large impact on performance.

6.2.2 Apache Spark Alternatives

The previous section analysed performance of Apache Spark in multiple scenarios, however, we should also investigate how that framework stacks up against other solutions. Therefore, this section details three comparisons between Apache Spark, Apache Hadoop, and a bash script based implementation. The experiments were performed on Microsoft Azure HDInsight cluster with 4 worker machines (as described in Section 6.2.0.3).

The first comparison included the fastest implementation the author created for each of the tools - Java was used with Hadoop and grep was used with the bash tools. Python was used with Spark, however, as shown in the previous section, all three languages supported by Spark offer similar performance for the benchmark job. The results are shown in Figure 6.6. Clearly, Apache Hadoop lags behind the "competition" as it is approximately two times slower than Spark and bash tools. In terms of runtime, a DataFrame-based Apache Spark implementation is surprisingly close to the bash tools which is an impressive feat considering that it offers fault-tolerance guarantees similar to Hadoop.

In software engineering, minimising the time it takes to develop software can often be as important as minimising runtime. Because Python reigns as the most popular programming language [37] and language proficiency aids fast development, the next comparison uses Python with the three solutions under the investigation. The Apache Hadoop job uses Hadoop Streaming, and the bash script uses Linux piping. Both PySpark jobs using RDDs and DataFrames were used here as well. The results are

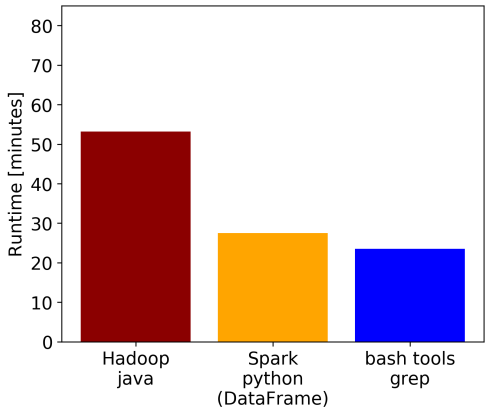

Figure 6.6: Comparison of benchmark job runtime between Hadoop (implemented in Java), Spark (implemented in Python and using the `DataFrame` API), and bash utilities (using grep).

shown in Figure 6.7. The runtime difference between the Spark job using DataFrames and the bash tools is very small but it should be noted that, as previously discussed, Spark is not really running the computation using Python. The Spark jobs using RDD API (which runs part of the computation in Python) was over two times slower than Hadoop.

Last but not least, workloads using regular expressions were used as well. The same Hadoop and bash tools implementations were used as in Section 6.1.5. Again, two implementations were tested for Spark - one using DataFrame's built-in `.rlike(regular_expression)` method and one using RDDs and a lambda function:

```
commonCrawlData
    .filter(lambda value: re.match('^WARC-Type: response', value))
    .count()
```

The results are shown in Figure 6.8. DataFrames-based Spark implementation outperformed even the bash tools. This could be thanks to lack of overheads related to Linux piping and inter-process communication, but it is also possible that the code running on JVM could be faster than Python for this use case. Once again, the job using RDDs is more than 2 times slower than the "competition".

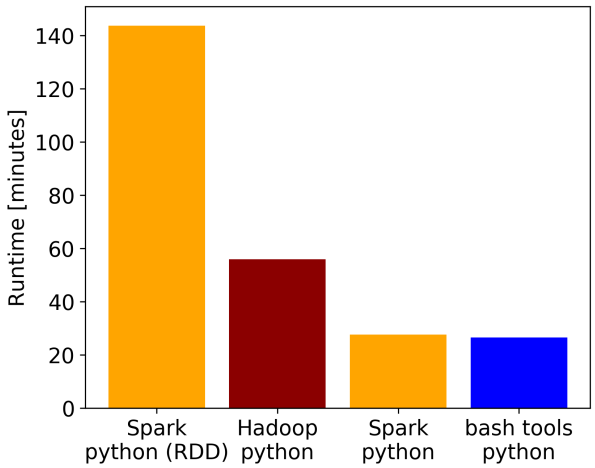

Figure 6.7: Runtime comparison between Hadoop (using Python and Hadoop Streaming), Spark (using `DataFrame` and `RDD` Python APIs) and bash tools (using Linux Piping).

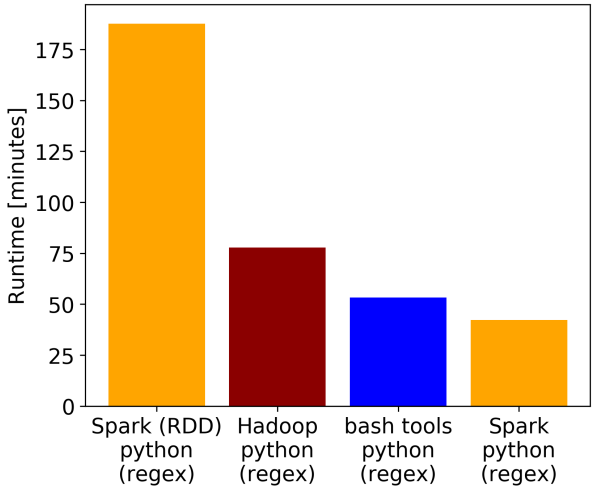

Figure 6.8: Comparison of Python benchmark job runtime (using regular expressions) between Hadoop (using Hadoop Streaming), Spark (which offers Python API), and bash tools (using Linux Piping)

Chapter 7

Wee Common Crawl Utility (`wecu`)

The results presented in previous chapters confirm that the alternatives to Apache Hadoop can offer better performance and more flexibility. However, this flexibility can be a double-edged sword as relatively small code changes can impact the runtime dramatically. Therefore, switching to a different framework (or a different programming language) is not always straightforward because an in-depth understanding of the framework's inner workings might be required to avoid "performance disasters" - when hundreds of terabytes of data need to be processed, even small performance issues can increase the runtime by hours or days. Moreover, Apache Hadoop might be chosen because of its popularity and familiarity, the abundance of related tutorials and books, because an existing codebase uses Hadoop, and so on.

However, a bash-tools based implementation can offer better hardware utilisation and reduce the runtime by as much as 70% compared to other frameworks. This leads us to a question: can we eat the cake (use existing, familiar Hadoop MapReduce code) and still have it (achieve runtimes comparable with Spark and bash tools)? A bash utility called **Wee Common Crawl Utility** (or `wecu` for short) is the author's attempt at doing just that.

The utility offers additional features such as automation of common and tedious tasks related to analysing Common Crawl data - for example, choosing a random sample of WARC files from some particular crawl. This section details the utility's features and some of the technical challenges the author faced during the implementation.

7.1 Command Line Interface

The Command Line Interface (CLI) itself was written in Python because it is a programming language very suitable for quick prototyping and development. However, the framework internally uses GNU parallel and `unpigz` to run the actual computation. A Python library called `argparse`, which has a number of convenient features, was used. Explanatory messages were provided where possible and `--help` or `-h` flags can be used to get a list of required and optional parameters, show explanations of those parameters, and so on.

7.2 Failure tolerance

A typical job processing Common Crawl will involve hundreds or thousands of tasks and it is not uncommon for some of them to fail due to programmer error, a network problem, a corrupted file, or some other issue. GNU parallel, used to distribute the computation among the worker machines, was configured to retry each task up to 3 times if a non-zero exit code is returned. When a tasks succeeds (i.e. returns a zero exit code) its output is sent to stdout, however, if the task fails 3 times the output of the last run will be sent to the stdout as well, so it's up to the programmer to handle task failures and minimise the risk of obtaining a garbled output. For example, this can be done by only printing to `stdout` after the task finishes processing the input data.

The author believes that machine failures are not going to be a major problem for most users because an average user is likely to be using just a handful of machines - for example, a small number of Virtual Machines running in a cloud environment. In this scenario, machine failures are unlikely because hardware problems are only a significant issue when a large number of commodity hardware machines are used over longer periods of time. Regardless, GNU parallel handles node failures "gracefully" by rerunning the affected tasks. This behaviour was tested by rebooting one of the worker machines while the computation was running and ensuring that the output of the job was unaffected.

7.3 Automated setup

7.3.1 Cluster configuration

Frameworks such as Hadoop and Spark are extremely complex and have many "knobs on the front" (i.e. many configuration parameters which can be tweaked). This gives the users a lot of flexibility, but it also means that cluster setup-up, configuration, and performance tuning are major pain-points. In terms of this flexibility versus ease of use trade-off, `wecu` was designed to be as simple to use as possible, and even the minimal configuration required was automated. If `wecu` is used on an Azure HDInsight cluster, the setup can be done using a single command:

```
wecu setup [cluster_password]
```

When running `wecu setup`, Hadoop configuration files are used to retrieve a list of worker machines. Next, `wecu` configures passwordless communication between the head node and the worker nodes by generating an ssh key and distributing it to the workers. Next, dependencies (e.g. GNU parallel, unpigz, sshpass) are installed on all machines. Last but not least, the number of logical cores available on worker machines is determined based on the `/proc/cpuinfo` Linux file.

After the setup is completed, `--check` flag can be used to verify that all required files are in place. It is also possible to check that all worker nodes are up using the `--check_nodes_up` flag.

7.3.2 Generating a sample of Common Crawl

The first step in analysing Common Crawl is obtaining a list of files which need to be processed. To do this manually, the user needs to 1) go to the Common Crawl website and choose a crawl, 2) locate the blog-post corresponding to that crawl, 3) download and decompress the file containing the filepaths, and then finally 4) prepend `s3://commoncrawl/` or `https://commoncrawl.s3.amazonaws.com/` to each line (depending on the protocol used). It is a simple but tedious process so the author decided to automate it. The following command starts a setup "wizard":

```
wecu generate-sample
```

After running the command, the user is presented with a list of all crawls available for download from AWS fetched from the Common Crawl Index Server [12] and asked to choose whether `warc`, `wat`, or `wet` files should be used, which crawl they should come from, and finally whether a full crawl, a sample of the first N files, or a random sample of N files should be used. The list of files is saved on disk and will be picked up by the framework the next time a job is executed.

7.3.3 Viewing configuration

After the configuration is complete, it is possible to view the list of chosen input files and worker machines. The command `wecu list machines` displays the list of worker machines, `wecu list files` displays the number of selected input files and the name of the crawl from which they come from (e.g. August 2019). Last but not least, `wecu list files --all` returns a full list of filenames.

7.4 Execution of arbitrary commands

Many tasks related to cluster management require running a number of bash commands on all worker machines, for example to measure hardware utilisation or to install dependencies. Doing this manually quickly becomes tedious, even with a small number of worker machines. `wecu` automates the processes as exemplified by the following commands:

```
(1) wecu execute "sudo apt-get install foobar"
(2) wecu execute "./setup.sh" --transfer_file setup.sh
```

Command (1) demonstrates that `wecu execute` takes a single argument - a string which can be an arbitrary shell command. If needed, the user can provide a list of files using the `--transfer-file` flag (see Command (2)). Those files will be distributed to all worker machines before executing the command.

7.4.1 Execution of MapReduce jobs

The following commands allow running MapReduce jobs written using any programming language and the reuse of existing Hadoop Streaming code:

```
wecu mapred ./mapper.py ./reducer.py
wecu mapred --jobs-per-worker 12 ./mapper.py ./reducer.py
```

The `wecu mapred` command takes two arguments: a path to an executable performing the map phase and an executable responsible for the reduce phase. The output of the map phase is sorted by key first and then by value. The communication between the executables is implemented using Linux Piping to avoid unnecessary disk accesses. By default, the number of concurrent tasks is set to the number of available logical cores, but the `--jobs-per-worker` flag can be used to change that number. Only one concurrent reduce task is supported in order to simplify the usage of the tool. In scan-and-count jobs, the output of the map phase is usually relatively small; thus, the runtime of the reduce phase is typically negligible so there is little benefit to allowing concurrent reduce tasks.

7.4.2 Scan-and-count jobs - without writing any code

The process of creating a scan-and-count Hadoop Streaming job using a high-level programming language like Python is straightforward but also time-consuming and error-prone. The user needs to create appropriate mapper/reducer files, set permissions to make them executable, implement the required logic, and so on. Even then, the whole effort might turn out to be fruitless if there is a typo in the code (Python is not a complied language so there are no compile-time errors to help us). The worst case scenario is that a reducer task failure causes the whole job to fail and the computation is wasted. For those reasons, the author decided to automate this process and allow running scan-and-count jobs without writing any code. The following commands can be used to do that:

```
(1) wecu sac "keyword"
(2) wecu sac --regex "^WARC-Type: response"
(3) wecu sac --regex \
             "^WARC-Type: response" \
             "^WARC-Type: request"
(4) wecu sac --regex \
             --by-file \
             "^WARC-Type: response" \
             "^WARC-Type: request"
```

```
(5) wecu sac --jobs-per-worker 8 "keyword"
```

Command (1) runs a simple MapReduce job which counts occurrences of the `keyword`. Command (2) counts the total number of matches for the regular expression passed as an argument (`^WARC-Type: response`). Multiple regular expressions (or search strings) can be used as well (see command (3)). Counting can be done per file, without aggregating the result by using the `--by-file` flag as exemplified by command (4). Last but not least, the number of concurrent tasks per worker machine can be manually adjusted using the `--jobs-per-worker` flag (see command (5)).

7.4.3 Monitoring hardware utilisation

Hardware utilisation can provide valuable insight into how efficiently the hardware is being used. Therefore, `wecu` allows quickly measuring and plotting CPU utilisation on all worker machines simultaneously. The following commands can be used:

```
(1) wecu utilisation graph_filename.png
(2) wecu utilisation --seconds 600 graph_filename.png
```

Command (1) can be used to monitor the CPU utilisation on all worker nodes. The data from each machine is then processed, transferred to the head node, and saved in a <`hostname`>`.usage.txt` file. By default, the utilisation is sampled every second over 2 minutes. The output files can be used for analysis directly, but wecu will also plot the results and save the graph in the location specified as a command line argument. For example, Figure 7.1 shows how the resulting graph will look like for a cluster with four worker machines. The number of data points to be collected can be changed using the `--seconds` flag. For example, command (2) will monitor utilisation for 600 seconds.

7.5 Performance

The runtimes achieved by `wecu` were expected to be very close to the results obtained using GNU parallel and `unpigz` because the tool uses those bash utilities to distribute the computation and decompress the input files. This was indeed the case - a scan-and-count job executed using `wecu sac` with two regular expressions outperformed the equivalent Hadoop Streaming job by approximately 30%.

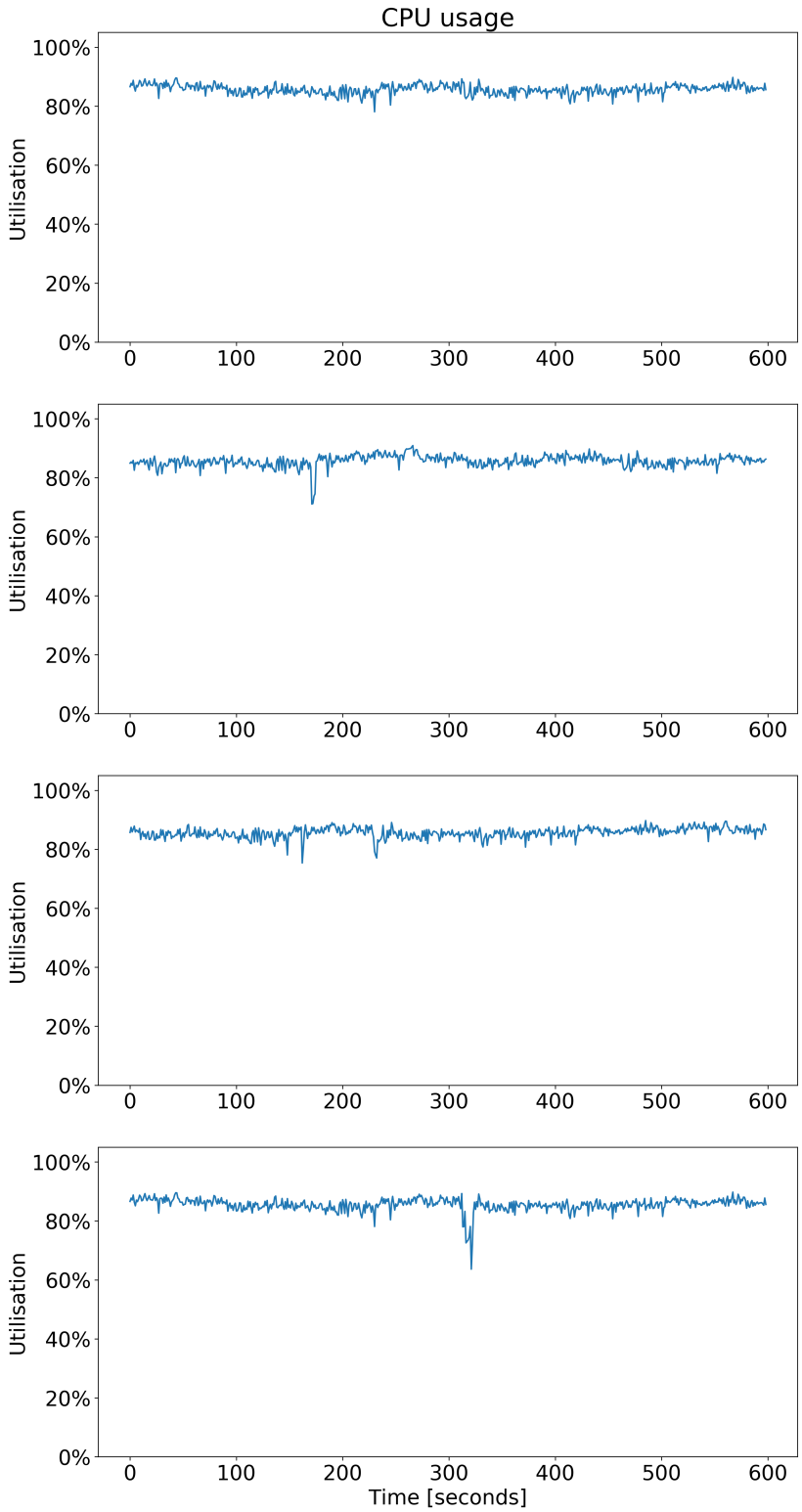

Figure 7.1: An example of what the CPU utilisation graph created by running `wecu utilisation --seconds 600` on a cluster with four worker machines looks like.

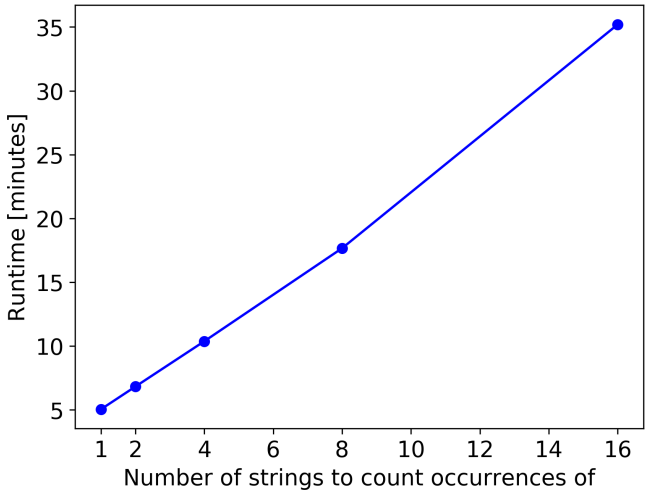

Figure 7.2: Performance of `wecu sac`: this graph shows the runtime (in minutes) on the y-axis and the number of search strings which were passed to `wecu sac` as arguments on the x-axis. The experiments utilised a smaller sample consisting of 100 WARC files.

The performance of `wecu sac` was tested with an increasing number of search strings passed to the utility as command line arguments as well. As the length of the strings used to run the tests (as well as the number of matches) can affect the runtime, the experiment was repeated 3 times, each time using a different set of 5 letter words from a list of random nouns generated using a popular online website[1]. The results are shown in Figure 7.2.

As expected, the graph shows that the runtime increases linearly with the number of input search strings.

7.6 Conclusions

The author believes that the results presented in this section prove that the utility can provide significant value to the users and hence Wee Common Crawl Utility was open-sourced and made available on GitHub[2].

The utility was "battle-tested" by the author - `wecu` was utilised to execute a number of scan-and-count jobs on the dataset to count the number of crawled pages belonging to a certain domain in order to establish whether Common Crawl was suitable to be

[1] `https://randomwordgenerator.com/`
[2] https://github.com/maestromusica/wecu

used as a source of data needed for one of the author's coursework projects. Because of the built-in automation features, the utility allowed performing the required analysis quickly and effortlessly, without writing any code.

Chapter 8

Future work

This report explored alternatives to Apache Hadoop such as using other frameworks (e.g. Apache Spark) or simply employing existing bash utilities to analyse the data. However, a number of other data processing frameworks, that are very popular but were not explored in this project, exist - for example Apache Flink [11]. Extending the analysis performed in this project to other frameworks would be very valuable to their users as it would allow estimating how significant the overheads introduced by those frameworks are.

Another interesting question that should be answered in the future is whether we can process the data faster using code written without the use of any Big Data frameworks or external utilities. The lack of communication overheads related to Linux piping or inter-process communication could, potentially, lead to improved performance. However, the implementation of such system from scratch is beyond the scope of this project because it would require implementing efficient file decompression, distribution of the computation, handling failures, and so on.

One of the major shortcomings of `wecu` is that the programmer needs to consider task failures and design the code such that they are handled appropriately. This unnecessarily increases the cognitive load on the user and makes errors more likely. Unfortunately, to the best of the author's knowledge, the way in which GNU parallel, the tool internally used to parallelise the computation, handles failures cannot be changed. Hence, a different utility would need to be chosen (or possibly implemented from scratch) to replace GNU parallel in order to address this problem.

Wee Common Crawl Utility automates a number of common, tedious tasks; however, there are a number of features which have not been implemented due to the time constraints of the project. The ability to produce a frequency distribution of the different strings matching a given regular expression would be useful. For example, computing the frequencies of the distinct strings matching `^Target-URI: ([a-z][^:]*)` regex would allow quickly computing the number of distinct crawled URLs, the number of duplicated pages, and so on. The tool could also allow only including pages matching a given condition in the search, for example, only the pages for which `Content-Type` header of the HTTP response has a particular value, or only those pages for which the response body contains some string. Last but not least, the development of the tool was focused on scan-and-count workloads, however, the utility could implement workloads of other types as well, such as extracting contents of pages matching some condition and saving them on disk (for example, in order to extract PDF files from the dataset).

Chapter 9

Discussion and Conclusions

Apache Hadoop is an immensely popular framework that is seen by many as the "default" choice for batch-processing large-scale data. In this project, an in-depth investigation into the framework's performance was conducted. A hypothesis, saying that a significant runtime reduction can be achieved for scan-and-count workloads by replacing Apache Hadoop with an alternative framework, was tested by comparing a number of data-processing solutions in a variety of settings and scenarios.

Firstly, it was shown that, when a single machine is used, implementations based on bash utilities (BTIs) outperformed Apache Hadoop by 35% to 70% and that Hadoop uses the hardware inefficiently. Similar trends were observed when HDInsight, a cluster with four worker machines hosted on Microsoft Azure, was used to perform comparisons between Apache Hadoop, Apache Spark, and BTIs. Hadoop was found to be slower than the alternatives in all tested scenarios and by a significant margin (with the exception of a Spark implementation using the RDD API).

Based on the results presented in this report, it is clear that Hadoop offers impressive scalablity, but its performance leaves a lot to be desired. The hypothesis tested in this project was confirmed - significant runtime reduction can be achieved by substituting Hadoop with an alternative framework as, with a few exceptions, Hadoop was outperformed by all tested alternatives, in all tested scenarios, and by a significant margin. However, an investigation into Apache Spark revealed that, in some circumstances, there are significant performance differences between the APIs offered by the framework. For example, the Python workload using the RDD API was approximately 5 times slower than the DataFrame implementation. Because of performance issues

like that, switching frameworks is not straightforward and users might still choose Apache Hadoop regardless of its performance. However, this leads not only to wasted computational resources but also wasted "programmer time" - creating scan-and-count MapReduce jobs is a simple but time consuming and error-prone process. The tool created by the author, called `wecu`, addresses this problem as it can be used to substitute Hadoop with almost no effort because it supports the MapReduce programming model and can run existing Hadoop Streaming code. The author believes that the tool can be of significant value to researchers working with Common Crawl because of its ease of use, efficiency, and convenient features, such as the ability to run scan-and-count jobs directly from the command line, without writing any code.

Bibliography

[1] pigz - Parallel gzip (Homepage). Retrieved: March 6, 2020, from https://manpages.ubuntu.com/manpages/disco/man1/pigz.1.html.

[2] Microsoft Azure. Virtual machine network bandwidth. Technical Documentation, 6 2019. Retrieved: March 6, 2020, from https://docs.microsoft.com/en-us/azure/virtual-network/virtual-machine-network-throughput.

[3] Ricardo A. Baeza-Yates, Carlos Castillo, and Efthimis N. Efthimiadis. Characterization of national web domains. *ACM Trans. Internet Techn.*, 7(2):9, 2007. Retrieved: September 23, 2018, from https://doi.org/10.1145/1239971.1239973.

[4] Bagmeet Behera. Blog Post, adsquare, August 2015. Retrieved: February 18, 2020, from https://www.adsquare.com/comparing-performance-of-spark-dataframes-api-to-spark-rdd/.

[5] Namrata HS Bamrah, BS Satpute, and Pramod Patil. Web forum crawling techniques. *International Journal of Computer Applications*, 85(17), 2014.

[6] Bagmeet Behera. Comparing performance of spark dataframes api to spark rdd, August 2015. Retrieved: February 18, 2020, from https://www.adsquare.com/comparing-performance-of-spark-dataframes-api-to-spark-rdd/.

[7] Miti S Bhat, Deepthi G Nair, Devyani Bansal, and J Vaishnavi. Data structure based performance evaluation of emerging technologiesa comparison of scala, ruby, groovy, and python. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–5. IEEE, 2012.

[8] Dhruba Borthakur. HDFS Architecture Guide. Technical Documentation, 08

2019. Retrieved: January 22, 2020, from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[9] Marc Brooker. Two traps in iostat: %util and svctm. Personal blog, July 2014. Retrieved: November 27, 2019, from http://brooker.co.za/blog/2014/07/04/iostat-pct.html.

[10] Mateusz Bukiewicz. Dive into PySpark. Technical Presentation, 2018. Retrieved: February 27, 2020, from https://www.slideshare.net/mateuszbuskiewicz/dive-into-pyspark.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[12] Common Crawl. CC Index Server. Technical Documentation. Retrieved: April 13, 2020, from https://index.commoncrawl.org/.

[13] Common Crawl. Common Crawl (Homepage), n. d. Retrieved: September 19, 2019, from http://commoncrawl.org.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] DigitalOcean. Running Cloud Native Applications on DigitalOcean Kubernetes, White Paper. Retrieved: November 12, 2019, from https://assets.digitalocean.com/white-papers/running-digitalocean-kubernetes.pdf.

[16] Vasileios S. Dimakopoulos. Apache Spark on Hadoop YARN Kubernetes for Scalable Physics Analysis. Technical Report, October 2018.

[17] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: the performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, pages 307–313. ACM, 2011.

[18] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[19] Microsoft Azure Documentation. General purpose virtual machine sizes. Retrieved: November 12, 2019, from https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general.

[20] Microsoft Azure Documentation. Hdinsight. Retrieved: November 12, 2019, from https://azure.microsoft.com/en-gb/services/hdinsight/.

[21] Lukasz Domanski. Is Common Crawl a reliable source of Web statistics? MInf Honours Project (Part 1), The University of Edinburgh, 2018.

[22] Adam Drake. Command-line tools can be 235x faster than your hadoop cluster, January 2014. Retrieved: November 16, 2019, from https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html.

[23] Satinder Bal Gupta. The issues and challenges with the web crawlers. *International Journal of Information Technology & Systems*, 1(1):1–10, 2012.

[24] Apache Hadoop. Homepage, n.d. Retrieved: April 4, 2019, from https://hadoop.apache.org/.

[25] Akaash Vishal Hazarika, G Jagadeesh Sai Raghu Ram, and Eeti Jain. Performance comparision of hadoop and spark engine. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 671–674. IEEE, 2017.

[26] Dominique Heger. Hadoop performance tuning-a pragmatic & iterative approach. *CMG Journal*, 4:97–113, 2013. Retrieved: November 29, 2019, from https://pdfs.semanticscholar.org/d0b7/300f5c1c8fbe3320c205c0fada555e8a729c.pdf.

[27] Holden Karau. Getting The Best Performance With PySpark. Spark Summit 2016 Conference Talk, IBM, June 2016. Retrieved: February 18, 2020, from https://www.slideshare.net/SparkSummit/getting-the-best-performance-with-pyspark.

[28] Juliet Hougland. Best Practices for running PySpark. Spark Summit Europe 2015 Conference Talk, 2015. Retrieved: February 27, 2020, from https://www.youtube.com/watch?v=cpEOV0GhiHU.

[29] Laurae (https://medium.com/Laurae2). Destroying the myth of number of threads

= number of physical cores. Medium.com post, April 2017. Retrieved: November 21, 2019, from https://medium.com/data-design/destroying-the-myth-of-number-of-threads-number-of-physical-cores-762ad3919880.

[30] Erwin Bolwidt (https://stackoverflow.com/users/981744/erwin bolwidt). Choosing optimal number of threads for parallel processing of data. Stack Overflow Stack Exchange. Retrieved: November 21, 2019, from https://stackoverflow.com/a/24149948/5817539.

[31] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 22–30. IEEE, 2014.

[32] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[33] Stephen Merity. Navigating the warc file format, April 2014. Retrieved: April 3, 2019, from http://commoncrawl.org/2014/04/navigating-the-warc-file-format/.

[34] Shengti Pana. The Performance Comparison of Hadoop and Spark. *Culminating Projects in Computer Science and Information Technology*, March 2016. Retrieved: November 15, 2019, from https://pdfs.semanticscholar.org/fab7/ef9a4d46d1af7ac5e26cf0146bf07decea05.pdf.

[35] Lucie Sadler. What is a VMware vCPU? Hyve Managed Hosting, July 2011. Retrieved: November 7, 2019, from https://www.hyve.com/what-is-a-vmware-vcpu/.

[36] Saeed Shahrivari. Beyond batch processing: towards real-time and streaming big data. *Computers*, 3(4):117–129, 2014.

[37] Stephen Cass. The Top Programming Languages 2019. Technical Article, IEEE Spectrum, 4 2019. Retrieved: March 25, 2020, from https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.

[38] O. Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[39] The Apache Software Foundation. Apache Hadoop 3.2.1. Technical Documentation. Retrieved: January 28, 2020, from `https://hadoop.apache.org/docs/stable/`.

[40] The Apache Software Foundation. Apache Spark 2.4.5 Documentation. Technical Documentation. Retrieved: February 18, 2020, from `https://spark.apache.org/docs/latest/index.html`.

[41] TutorialKart. Blog Post, n.d. Retrieved: February 22, 2020, from `https://www.tutorialkart.com/apache-spark/dag-and-physical-execution-plan/`.

[42] Rahul Vaghasia. What is a VMware vCPU? accu Web Hosting, January 2018. Retrieved: November 7, 2019, from `https://www.accuwebhosting.com/blog/what-is-virtual-processor-or-vcpu/`.

[43] 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.

[44] W3C. The World Wide Web Consortium (W3C), W3C homepage, 2018. Retrieved: April 4, 2019, from `https://www.w3.org/WWW/`.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[46] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. Retrieved: November 12, 2019, from `https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf`.

Appendix A

A.1 Apache Hadoop configuration

Configuration parameter name	Value
Scheduler	`CapacityScheduler`
`yarn.nodemanager.resource.memory-mb`	64309MB (all available RAM)
`yarn.scheduler.minimum-allocation-mb`	512MB
`yarn.scheduler.maximum-allocation-mb`	64309MB (all available RAM)
`yarn.scheduler.capacity.maximum-am-resource-percent`	10%
`mapreduce.map.memory.mb`	512MB
`mapreduce.tasktracker.map.tasks.maximum`	32 or 64
`mapreduce.tasktracker.reduce.tasks.maximum`	1

Table A.1: Most important Hadoop configuration parameters and their chosen values