UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

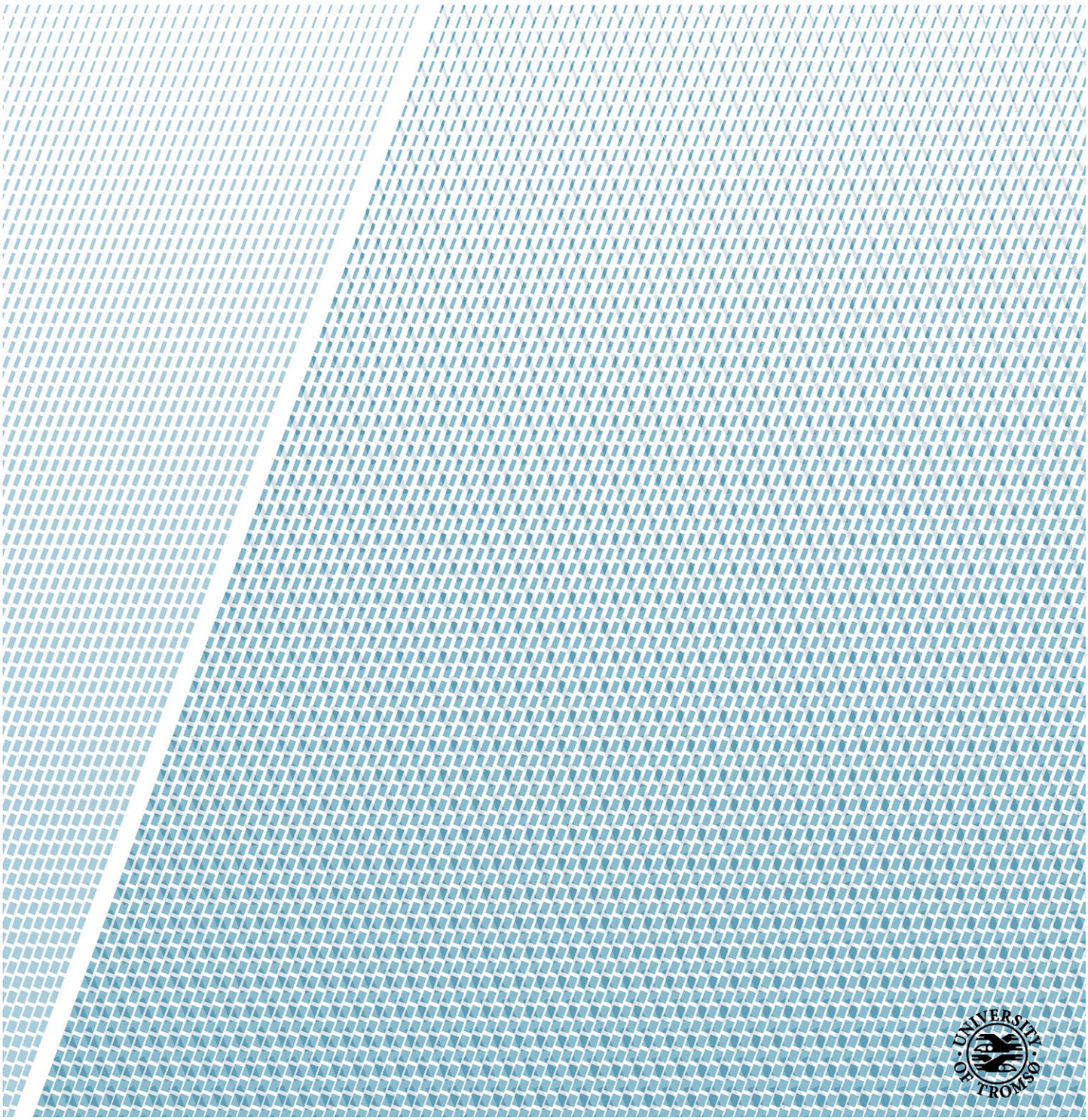
Big Data Facilitation and Management

A requirements analysis and initial evaluation of a big biological data processing service

—

Jarl Fagerli

INF-3983 Capstone Project in Computer Science - December 2015



Abstract

Biology is now considered a big data science, often attributed to the advent of next-generation sequencing technologies producing large amounts of biological data. In spite of this, most biological data analyses are done on relatively small datasets, as a result of the typical biologist neither having the resources nor the technical proficiency to process and manage large-scale data. Recognizing this, our project provides a requirements analysis and an initial evaluation for an approach to data management and processing of big biological data. We identified the following core requirements: 1) Large compute resources are needed for analysis purposes; 2) software expressly designed for big data processing is essential; 3) the software frameworks should ameliorate data management; and 4) external storage solutions are likely necessitated. Based on the requirements listed in the prior, we: Pertaining to 1) and 4), performed an exploration of a cloud computing service and its associated cost, including a comparison with local resources to showcase another typical computing environment; moreover we found that the frameworks Spark in combination with Hadoop alleviate both requirements 2) and 3).

We found, by comparison, that: usage is similar; using a cloud service for both compute and storage resources incurs overhead of remote data retrieval; data retrieval throughput scales with the amount of tasks deployed viz., the number of virtual cores available to Spark; allotment estimation of cluster resources is non-trivial; and using a cloud service provider is more expensive than buying a cluster to own, given our extraordinary scenario. We also performed an initial evaluation of an interactive biological data analysis method, k-mer distribution, using it to identify issues of interactive data cleaning, showing that optimizing k-mer distribution is not suitable for an interactive application due to elongated execution times, averaging almost three minutes. Spark was found easy to use and provides near endless tweaking and tuning possibilities.

Whether to buy to own or rent resources comes down to trade-offs regarding cost and the synergistic performance characteristics of the different entities within the service's architecture. Cost favors buying a cluster to own, based on our special scenario. Using a cloud cluster, hardware failures are easily fixed by provisioning another node; the local cluster gives more control and maintenance responsibilities, and longer restoration times in case of hardware failures. Performance needs to be evaluated in context of the other architectural components of the system, e.g., by measuring inter-service communication performance, availability and reliability requirements, and physical locations of the different services. The nucleotide representation of k-mers needs to be revisited, and an efficient implementation should be prioritized before attempting to increase cluster size.

Acknowledgements

I would like to express my gratitude to my advisor, Associate Professor Lars Ailo Bongo, for his continued guidance and support throughout the course of this work.

I would also like to extend my thanks to my co-advisors Inge Alexander Raknes and Giacomo Tartari for their technical and research related assistance.

Appreciation is also extended to Jon Ivar Kristiansen for sharing his insights in computer clusters.

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

API application programming interface.

ASF Apache Software Foundation.

AWS Amazon Web Services.

BDAS Berkeley Data Analysis Stack.

CPU central processing unit.

DAAS Data as a Service.

DAG directed acyclic graph.

DFS distributed file system.

DNA deoxyribonucleic acid.

DRAM dynamic random-access memory.

EBI European Bioinformatics Institute.

EC2 Elastic Compute Cloud.

EMBL European Molecular Biology Laboratory.

EMR Elastic MapReduce.

ENA European Nucleotide Archive.

GPS Global Positioning System.

GPU graphics processing unit.

GUI graphical user interface.

HDD hard disk drive.

HDFS Hadoop Distributed File System.

HPC high-performance computing.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

LRU least recently used.

MPI Message Passing Interface.

NAS network-attached storage.

NIC network interface controller.

NVM Non-Volatile Memory.

ORC Optimized Row Columnar.

PCI Peripheral Component Interconnect.

RDD resilient distributed dataset.

RDG resilient distributed graph.

S3 Simple Storage Service.

SSD solid-state drive.

SSH Secure Shell.

VPC virtual private cloud.

WARC Web ARChive.

YARN Yet Another Resource Negotiator.

Chapter 1

Introduction

Since the inception of high-throughput genomic sequencing, biology has become one of the big data sciences, and progressively more labs are able to generate large quantities of data as sequencing equipment is becoming more affordable. Figure 1.1 shows a graph of the contents of the European Molecular Biology Laboratory (EMBL)-European Bioinformatics Institute (EBI) [16] data platforms and their growth trends over the past few years. The EBI stored 20 petabytes of biological data and back-ups as of December 2014 [44], and is one of the largest data repositories of its kind in the world. Note that raw sequencing data was the largest data platform as of 2014. Thence, as a result of the increased data growth, biologists are now confronted with massive amounts of data and the issues and benefits that it entails [53]. Many biologists are often limited by a lack of knowledge with regards to handling big data and have limited experience with big data frameworks or systems, and if they are familiar with such systems, the needed compute resources are not always readily available for use. Even when having both the proficiency and resources, complex engineering problems still remains an issue that must be addressed, like infrastructure needs, the limitations of biological analysis tools used, and extensive knowledge of hardware and software.

Big data also carries great potential for gaining new knowledge. Processing of big biological data can help discover new drugs, find new viruses in different species [53], and gain information about illnesses, diseases and provide better healthcare [62]. It can even evolve to be used in clinical practice, given acceptable analysis speed and methods. A big biological data analysis service is therefore needed.

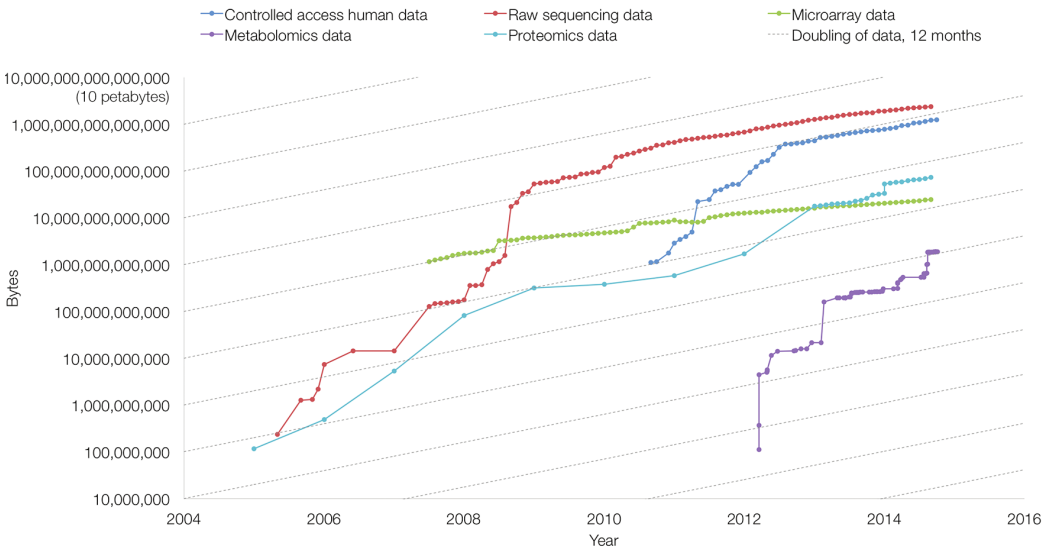


Figure 1.1: EMBL-EBI data growth by platform. Derived from [15]

The current big data analysis trend is settled on cloud-based solutions [53, 40], allowing "anyone" access to compute power without the need to buy and maintain hardware, through ad hoc creation and configuration of clusters in which only the resources used. One of the principal ideas is to have all data in the cloud, including the tools and compute power needed to do analyses on data from a multitude of sources. Having data readily available in a cloud may also promote sharing. The cloud is also a neat abstraction for people unfamiliar with computer science, and a nice graphical user interface (GUI) can help eradicate concerns researchers might have about using new systems, which in turn can broaden the user base. Additionally, some large biological datasets are already available as Data as a Service (DAAS) in the cloud, like 1000 Genomes[1] and 3000 Rice Genome[2] on Amazon, which can be another motivating factor for embracing the cloud.

In this work we determine – based on our operational scenario – what physical resources are needed to run a big biological data analysis service; deduce which software frameworks and infrastructures that best fit said service; and establish the expected expenditures of renting cloud-based resources, as opposed to purchasing resources to own, for a big biological data analysis service.

[1] `http://aws.amazon.com/1000genomes/`
[2] `https://aws.amazon.com/public-data-sets/3000-rice-genome/`

This project provides an analysis of requirements for, and an initial evaluation of, an approach to big biological data processing and management on computer clusters. We identified the core requirements for such a system: The necessary compute power is an obvious, but indispensable, requirement. Secondly, a processing framework for big data is necessary, particularly one good at data-intensive tasks is favored. Thirdly, a rich set of operations easing data management should be supported by the framework, advantageously one that interfaces with the widely used Hadoop stack. Lastly, storage for input and output data needs to be facilitated. Depending on the cluster used, it is probable the local Hadoop Distributed File System (HDFS) [63] will not suffice, necessitating some kind of external storage repository.

Inferred from the requirements, a computer cluster of some size is required (typically the larger the better) as well as triple-digit tera-scale storage, depending on the size of the data to be processed. Moreover, we propose to apply the Spark [71, 70] framework on top of Hadoop as a good fit for both processing and distributed data management. It is the current leading edge of big data processing, is built to integrate well with Hadoop [30], and is in active development.

In order to display characteristics of the different environments, ease of use, and an estimation of cost using an Amazon Web Services (AWS) solution for processing of big biological data, experiments were conducted using both a local cluster with input data stored in HDFS as well as an AWS Elastic MapReduce (EMR) cluster interacting with Simple Storage Service (S3) for input. The experiments encompassed extracting links from Web ARChive (WARC) files and running PageRank on these. PageRank is a widely known algorithm that, in its breadth, also is representative for a big biological data service, and with input available in S3, as well as experiments using an interactive data cleaning method, k-mer distribution, to ascertain whether or not interactive optimization is practicable.

For large datasets, our experiments showed network characteristics of interaction between EMR and S3, finding it to slow down execution times when needing to pull large amounts of data, which is due to limitations in network throughput. By adding more nodes, the throughput would increase, and, although not financially practical, the best case would be to have one task per file to download. As expected, we found EMR to be a lot more expensive at an hourly rate than buying hardware to own locally, if used continuously, with the cheapest pricing option being about double that of using a local cluster in our circumstances. We identified some optimizations that should

be looked into and thus serve as future work, notably serialization methods, compression formats, and file formats. Our lessons learned by using the two different environments are quite similar in terms of user interaction, although different tuning options in Spark are required when dealing with different hardware and software versions, as well as needing to account for the resources needed for stable storage when dynamic random-access memory (DRAM) is exhausted. Additionally, we performed data cleaning experiments by implementing the k-mer distribution method in order to establish issues and challenges germane to interactive data cleaning. The results showed that interactively doing k-mer distribution optimization is infeasible as the average time it takes for one execution is too large, making a client wait for too long for the next interaction.

1.1 Big Data

As previously mentioned, biology has become a big data science. Big data exceeds the capacity and capabilities of traditional database systems and is defined by three distinguishing factors: volume, velocity, and variety. Volume refers to the large scale of the data, its sheer vastness; velocity refers to the speed at which data is created; and variety refers to all the different types data can take the form of [55]. This has introduced a great many issues related to the hardware and software needed to handle this unwieldy data, even challenges related to the requirement of specialized personnel to interpret the results following the analysis of the big data.

In commercial settings, correctly using the results has the potential of gaining competitive advantages. In science, processing of big data holds promise of providing a lot of knowledge if interpreted correctly. A prominent example is genomics, which can be used by healthcare facilities, drug firms, and biotechnology companies for research purposes [53].

1.2 Bioinformatics

Bioinformatics is the application of computer science methods to molecular biology data: the application of computational techniques to analyze, organize, and make sense of (large) biological data [51]. It is an interdisciplinary field in that it combines several other scientific fields including mathematics,

statistics, and engineering, as well as the aforementioned.

Big data management is one of the important aspects that bioinformatics are currently seeking to address. The recent advances in high throughput technologies has allowed biologists to create massive amounts of biological and clinical data, which development is chiefly attributed to the advent of next-generation sequencing technologies. The rate of growth and accumulation of this data is hard to handle, and the data is costly to store and analyze.

There has been a lot of research done on the field of developing algorithms that can extract useful information from these enormous datasets. Examples from the Raphael Lab of Brown University [24] include: THetA2 [60] (an improvement of THetA [58]), short for Tumor Heterogeneity Analysis, using high-throughput sequencing data to estimate tumor purity and copy number aberrations (see §4.5 for more information). PREGO [59] (Paired-End Reconstruction of Genome Organization) which uses paired-end sequencing data to reconstruct cancer genomes (refer to §4.6 for more information). GASV and GASVPro [64] (Geometric Analysis of Structural Variants) uses paired-end sequencing data to analyze structural variation in normal and cancer genomes.

1.3 Organization

The remainder of the document is organized as follows: Chapter 2 covers a requirements analysis, describing an operational scenario, elaborating on the requirements of a big biological data processing system, and discussing different architectonic approaches, infrastructures, and platforms; Chapter 3 presents an initial evaluation of the different systems mentioned earlier through experiments; Chapter 4 contains some background information associated with the frameworks, methods, and nomenclature used; Chapter 5 concludes.

Chapter 2

Requirements Analysis

Our ultimate goal is to provide a generalized big biological data analysis service with which a user can provide a script containing an algorithm to be run, and that the service then transparently takes care of the rest and efficiently produces a result of the computation specified in the script. This document will however be limited to the scope of a single use case, i.e., a separate task.

2.1 Operational Scenario

Data cleaning is a common first step when handling big data, and the data cleaning steps are in most cases domain specific. One data cleaning method used in biology is optimizing the k-mer distribution.

The operational scenario for which requirements are analyzed is that of an analysis using raw deoxyribonucleic acid (DNA) sequencing data from high-throughput sequencing machines. More specifically, one use case is interactively computing the k-mer distribution, i.e., find an optimal – or satisfactory – k-mer distribution value by trimming nucleotides from the start (head) and end (tail) of a sequence by tweaking parameters. The k-mer distribution can tell the biologist with what cut-offs from the head and tail provides the optimal distribution value, which is done in preparation for other further analyses.

The idea is that a biologist is presented with a nice Web interface which

can be interacted with by first specifying a file to be analyzed. The analysis consists of first computing the k-mer distribution of the file given a value k. Once the initial computation is done, the biologist may choose, based on the resulting value, whether to tweak more, or to end the analysis having found satisfactory parameters for the head and tail cut-off values. The intermediate results of the k-mer distribution should be computed in reasonable time, as slow interactive systems tend to make users lose interest and will either impact their work negatively or make them not bother at all. The workflow as described is visualized in Figure 2.1, in the tweak step the head and tail values are set, and the user can recompute and tweak as many times as needed.

This concept can easily be expanded upon to include many other analysis and data cleaning algorithms that uses raw sequence data and Phred quality scores.

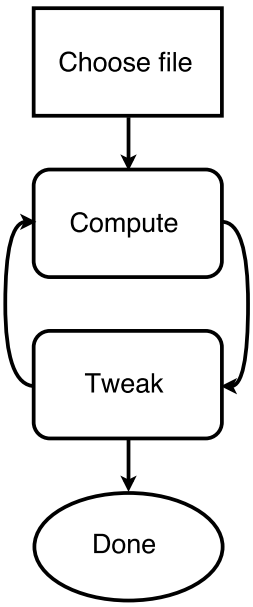


Figure 2.1: General workflow

2.2 Architecture

The general architectural traits imposed are the necessary resources needed to complete a task such as the operational scenario in reasonable time and to have access to – and storage space for – big biological data at large (global or external) repositories. Thus, the envisioned architecture will need compute resources in the form of a computer cluster, and favorably one built for data-intensive computing with a great amount of DRAM. The storage repository needs to be large, probably tens to hundreds of terabytes, and must provide a compatible interface and protocol for the cluster and the used framework to be able pull data to be processed from it.

The larger repositories are in most cases not in-house, and as such results in considerably higher latencies and lower throughputs than the local distributed file system (DFS) and DRAM. The global storage repository is generally assumed to be smaller than the external storage repository. Examples of global storage repositories are S3 (§2.3.3) and NorStore (§2.4.3), and a typical example of an external storage repository is the European Nucleotide Archive (ENA) [17].

For the cluster to be able to work on and process data, it needs access to said data. There is a natural intra-cluster storage hierarchy consisting of DRAM and DFS, of which DRAM provides much faster response times than DFS, but in return is also a lot smaller in terms of volume; this is reflected in Figure 2.2 within the stipulated box.

The proposed architecture of the storage hierarchy is depicted in Figure 2.2, showing the main components and illustrating that latency and size increases from the top down. The hierarchy will work like a multilevel cache for the datasets that are to be processed, such that only when a miss on a dataset occurs in DRAM will DFS be searched for the dataset, and only if a miss occurs in DFS will the global storage repository be searched, and finally if another miss occurs, the dataset will be retrieved from the external storage repository if present. One could choose to start working on the data immediately after pulling it, leaving it only in DRAM, or one could save it to DFS before continuing the analysis process. Additionally, depending on the level of privileges granted at the global repository, it can be used to remotely pull and store data directly from the external repository through some exposed application programming interface (API) higher up the hierarchy. If this is not possible, the data needs to be manually stored at the global storage repository after being pulled to DRAM.

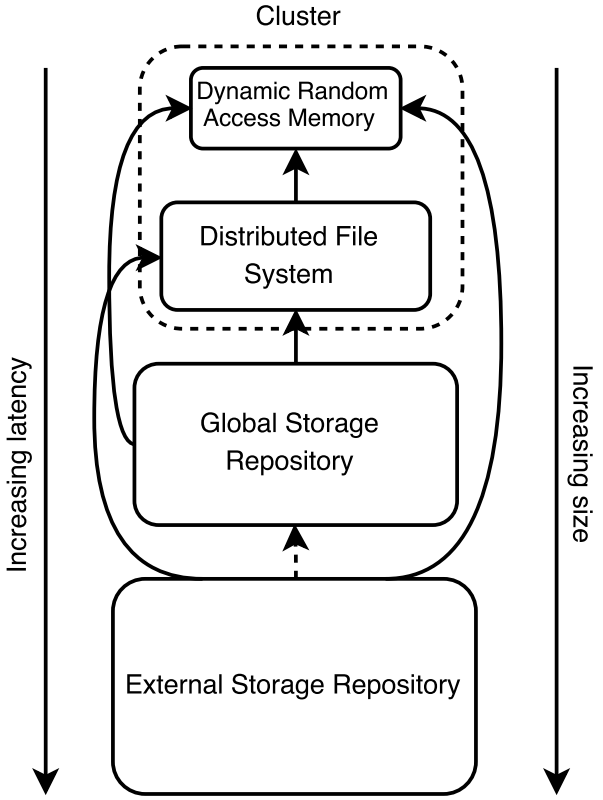


Figure 2.2: General architectonic storage hierarchy

2.3 Platforms

To realize the proposed architecture, we consider different approaches and platforms. The following describes the different software platforms used, and considered, throughout this work. It includes characterizations of distributed frameworks for storing and processing large scale data, as well as platforms suited to run these frameworks.

2.3.1 Hadoop

Hadoop is an open-source framework designed to be run on large Linux clusters composed of machines using commodity hardware. It is designed for distributed data analysis at scale in a reliable fashion by providing fault tol-

erance and high availability. There are four modules included in Hadoop: Hadoop Common, HDFS, Hadoop Yet Another Resource Negotiator (YARN), and Hadoop MapReduce (see §§4.1.1 and 4.1.2 for brief descriptions). Our project makes use of HDFS and YARN for storage, scheduling, and resource management needs.

2.3.2 Spark

A processing engine is required to deal with the large amounts of data, both in terms of the processing itself, but also for data handling. It should support the established Hadoop framework and be applicable to both compute and data intensive tasks as well as rich support for additional tools.

Spark is a framework built to interoperate with Hadoop, and is by many considered the state of the art. It is a generalization of the MapReduce system (§4.1.1, supporting the implementation of several other programming models in addition. The main advantage it brings over Hadoop MapReduce is that it seeks to fully utilize the DRAM of each node rather than only using disk storage, which can significantly increase performance of I/O-intensive jobs and iterative computations. Spark has been shown to outperform Hadoop MapReduce by up to 20× in iterative applications [70] and to be efficient at shuffling data, as evidenced by its 100 TB GraySort benchmark [67]. The speedup is mainly a result of having data to be accessed readily available in DRAM and the ability to keep unserialized objects in memory, diminishing both I/O and (de-)serialization costs.

Having been adopted by Cloudera [7] and AWS [21], it seems reasonable to argue that Spark is a platform mature enough to be chosen, however, it does not yet have the amount of support infrastructure that Hadoop has acquired over time. Spark is in active development and had more than 700 contributors at the time of this writing [19], showing no indication of declining popularity. With increasing functionality, optimizations and the beneficial in-memory computing with unserialized resilient distributed datasets (RDDs) the adoption of Spark is deemed worthwhile.

Spark interfaces with Python, Java, and Scala. Being dynamically typed, Python is likely the slowest option in terms of performance, but if Python is doing little processing and for the most part uses Spark libraries, it might be negligible. As Spark is built using Scala, it is probably the language that provides the best integration and native support. Additionally, Scala can

seamlessly make use of Java libraries, thus for the aforesaid reasons, it was chosen as the language for this project.

For more information on Spark refer to §4.1.2.

2.3.3 Amazon Web Services

A cluster is required to run Spark and Hadoop and do the actual computations and facilitate disk storage. AWS, launched in 2006, is Amazon's cloud-computing (technology and content) platform offering different services for computing, storage, and more. It provides an infrastructure that is robust, scalable, reliable, and inexpensive [36]. The services most pertinent to this work are EMR[1] – which is built on top of the Elastic Compute Cloud (EC2)[2] – and S3[3].

EC2 is a compute service that provides quick and configurable capacity of computing resources and several different types of servers, e.g., specialized servers for data-intensive and compute-intensive applications. The instances are chosen and configured to the user's specifications and is launched within short time of creation, and are also able to be resized.

S3 is a storage service that provides durable and scalable object storage, providing different types of storage classes. The storage classes are primarily based on data access patterns, for example one designed for frequently accessed data and another for long-term archiving of data. Data is by default redundantly stored in multiple facilities across multiple devices for most of the storage classes, but the level of redundancy may be configured by the client.

EMR manages and sets up a Hadoop cluster using EC2 instances to distribute and compute data. It can interact with S3, among others, for storage needs, but also supports interaction with Amazon DynamoDB, in addition to the provided HDFS-storage. It can also be configured to set up frameworks, including Spark and Presto. Only the amount of resources used across the services are paid for, the customer does not need to worry about handling different failure scenarios, and the instances are launched in a virtual private cloud (VPC) environment, which lets the user control the network topology

[1] `https://aws.amazon.com/elasticmapreduce/`
[2] `https://aws.amazon.com/ec2/`
[3] `https://aws.amazon.com/s3/`

and manage traffic routing and filtering [34]. Spark uses the Apache Hadoop cluster manager YARN [66] in AWS EMR instances.

2.4 Infrastructures

Infrastructure needs can be met in several different ways. The most common approaches are acquiring resources through cloud-based solutions or buying the resources, in the form of hardware, to own. In a cloud environment, clusters may be created and configured ad hoc through some API or GUI made available by the cloud service provider, allowing for elasticity and easy configuration. Locally owned resources must be set up, configured, and tuned manually, and are not nearly as elastic as cloud-based clusters. Additionally, the tasks of setting up and configuring a cluster are not trivial, requiring a deep understanding of hardware, software, and software tuning to best utilize the hardware characteristics.

Common to the approaches is the way the service needs to be set up and started, which is done using some network protocol like Secure Shell (SSH) for data copying, initiating program execution, and other managerial operations.

Our use case requires that the clusters are configured with the Hadoop and Spark frameworks: the former for HDFS and YARN; the latter serves as the processing engine and interfaces with the Hadoop framework.

An important design choice that must be considered with respect to several factors is whether to purchase, and thus own, hardware to be locally available, or to pay for compute and storage services from a commercial provider. Cost-efficiency is the gist of this matter. The cost-performance trade-off needs to be evaluated with regards to the expenses of using a cloud service in order to reach a conclusion on whether or not it is feasible to rent compute and storage resources from a commercial cloud service provider (e.g., AWS), or to buy a new cluster and have the hardware at hand. Using a cloud-based solution, the cost is distributed over time and only the amount of resources used is paid for; buying a cluster to own privately means there will one large initial sum and then only maintenance costs, which includes power consumption.

The following subsections elaborates on examples that include a bought cluster that is locally available, proposes an AWS EMR cluster configuration, and a supercomputer.

2.4.1 Amazon Web Services

Services like AWS can supply a practically unlimited amount of compute and storage resources, but it comes at a price. There are several EC2 instance types, i.e., hardware configurations, available: general purpose, compute optimized, memory optimized, graphics processing unit (GPU) optimized, and storage optimized [10]. The best fit instance type for our purpose is likely one with a lot of memory, as Spark considerably benefits from being able to have as much of the working as possible set fit in DRAM, but in general this of course depends on the nature of the application to be used.

An example cluster configuration is specified using the M3, general purpose, EC2 instances. The instance type used is the `m3.2xlarge`, the configuration of which is listed in Table 2.1. The central processing unit (CPU) listed in said table has 10 cores with a base frequency of 2.5 GHz, 20 threads and 25 MB cache; however only eight threads (logical cores) are assigned to each `m3.2xlarge` instance [10].

Table 2.1: Hardware specifications of an `m3.2xlarge` instance [10]

CPU	Intel® Xeon® Processor E5-2670 v2 (25M Cache, 2.50 GHz)
Memory	32 GB
Storage	2 × 80 GB
Operating system	Amazon Linux AMI (2015.03)
Cluster manager	YARN
Approximate cost	$2,300 (in Norway)

Both Spark papers', GraphX', and MLlib's experimental setups used the general purpose `m`-family EC2 instance types (`m1.xlarge`, `m2.4xlarge`, `m3-.2xlarge`) [71, 70, 68, 56], while in the Spark SQL paper, storage optimized `i2.xlarge` instances were used [33]. Neither paper reason about their choice of instance types, but it is plausible that the `m`-family instance types are chosen because they represent the typical case cluster computer hardware composition with respect to the resource balance, for breadth. In furtherance of utilizing DRAM to support in-memory computations and requiring a balanced amount of CPU power, the `m3.2xlarge` instance type was chosen for this work.

It is also worth noting that the local storage of the `m3.2xlarge` instances are solid-state drives (SSDs), providing much faster read and write speeds than conventional hard disk drives (HDDs) with rotating disks, but are in return

more expensive per byte of storage. This is advantageous for the performance in the case of a lot of I/O operations, like reading input from disk or writing results to disk, and also for intermediate results when Spark needs to spill data to disk upon depleting DRAM, which is commonplace when operating with large datasets. If all data fits in DRAM it will not make too much of a difference, only impacting input reading speed (if read from disk) and output writing speed.

An advantage of using paid services is that failure scenarios need not be taken into consideration, being a problem on the service provider's end to enforce availability. Upon encountering a broken node, it is automatically removed from the virtual cluster and a new node to be added is provisioned. However, driver failures will still pose a problem, as there are currently no driver failure recovery mechanism in Spark, in contrast to Spark Streaming which periodically persists its state for recovery purposes by checkpointing. In addition to this, an application that scales linearly with added hardware can benefit from adding instances and completing work in a shorter amount of time. Because only the amount of hours used is paid for this may be leveraged to work in the client's favor.

In order to get a roughly comparable cluster configuration to the locally owned cluster at hand, the example AWS EMR cluster consists of of 10 $\times$ `m3.2xlarge` instances, to get an approximately equal amount of memory and CPU resources. Assuming an I/O bound application, this approximation places little emphasis on balancing storage, but the clusters will be tantamount in terms of memory and CPU resources. If need be, additional instances can be added for scaling purposes without the need to manually install and tune the hardware.

The lack of storage space available on the EMR cluster itself needs to be addressed, and supplemental storage from S3 is a compatible solution and the logical choice when already using the AWS platform. S3 has defined different storage classes principally based on how frequently the stored data is accessed. The different storage classes are S3 Standard (general purpose), S3 Standard - Infrequent Access, and Amazon Glacier (archive) [12], and presuming a moderate amount of access requests, the reasonable choice is likely the S3 Standard storage option.

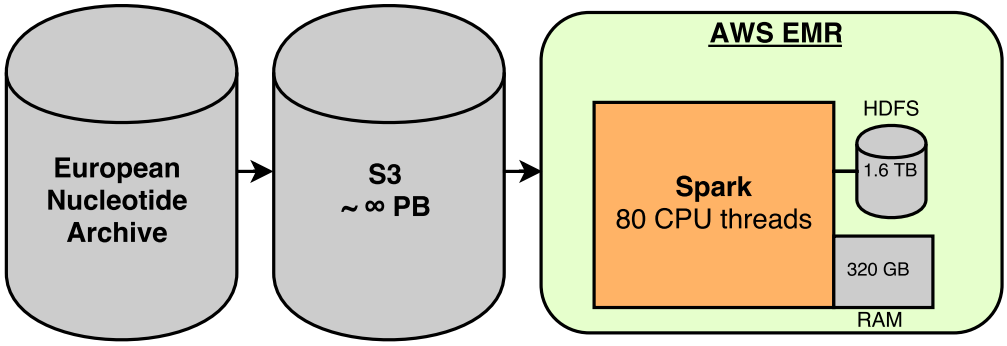


Figure 2.3: AWS EMR, S3, and ENA stack. Numbers are aggregated specs of 10 × `m3.2xlarge` nodes

2.4.2 `ice2`

`ice2` is a 10 node cluster owned by the Center for Bioinformatics (SfB) [13] and was composed with data-intensive computing in mind. It will serve as the example of a bought cluster, and is available for use. The expenses spent on computing resources might be diminished if used contrary to using rented resources, if used continuously over a long period of time. The setup per node in the cluster is listed in Table 2.3. Each node's CPU has 4 cores with a base frequency of 3.6 GHz, 8 threads, and 10 MB cache; in total, the cluster consists of 40 CPU cores, 320 GB DRAM, and 40 TB of storage.

Table 2.2: Hardware specifications of an `ice2` node

CPU	Intel® Xeon® Processor E5-1620 (10M Cache, 3.60 GHz)
Memory	32 GB
Storage	4 TB
Operating system	CentOS Final (6.7)
Cluster manager	YARN

Scaling by adding hardware – although not probable – is a possibility, yet might be counterproductive when trying to minimize cost. There is also the risk of ending up with heterogeneous nodes, as it might be hard to obtain identical hardware to the nodes currently in the cluster, which could pose a problem as time goes by. Additionally, hardware added needs to be set up and the cluster tuned to reflect the changes made, which can be time-

consuming. Furthermore, failure scenarios needs to be handled locally by qualified personnel when using private equipment. Typically, replacement of hardware upon failure is covered by a warranty from the cluster hardware provider, but it will require time to physically replace the broken hardware. A one-day turnaround for replacement is common, but if the incident happens before the weekend or before a holiday, the turnaround time increases.

The limited amount of storage will be an issue when the size of the datasets grow larger, and it is worth noting that the storage on hardware level are hard disk drives with rotating disks and read-and-write heads yielding slow read and write speeds. The limited raw storage capacity of the cluster is likely to require other, external, storage facilities like NorStore – which also is available for use – to work as a repository for input, and output if need be, data for the application.

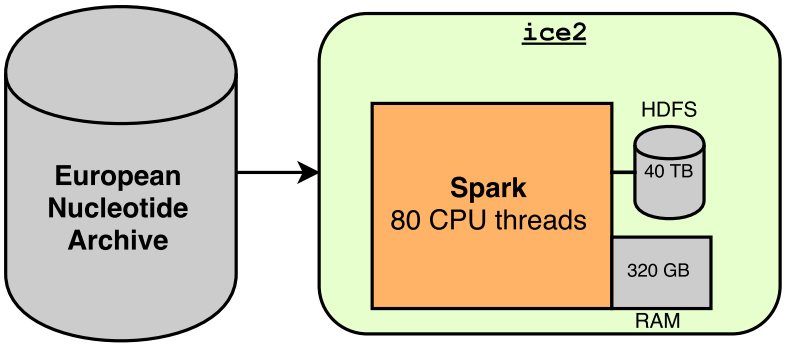


Figure 2.4: The `ice2` and ENA stack. Numbers are aggregated specs of all 10 nodes

2.4.3 NorStore

An alternative to using Amazon's storage services is using the research infrastructures such as Stallo in combination with NorStore [22]. NorStore is a Norwegian platform that offers storing services for scientific data to all scientific disciplines that are having large scale data storage needs. It enables research and education in cooperation with Notur through mediating the creation and sharing of data at scale. Data is either stored in a Project Area or in an Archive: The Project Area should contain data that is actively used for processing and analysis, while the Archive is for long-term storage when data is no longer manipulated. It is common that data stored in the

Archive is made publicly accessible.

Their hardware resources consists of 8 pebibytes of storage located at the University of Oslo with a mirror at the University of Bergen. Approximately half is tape storage while the other half resides on disks. 0.5 pebibytes of disk storage is located at the University of Tromsø, and is available for use.

2.4.4 Stallo

Stallo [1, 27] is a supercomputer located at the University of Tromsø. It is part of the Notur Norwegian academic high-performance computing (HPC) infrastructure and facilitates parallel, serial and large I/O jobs. Stallo is intended for computations not requiring a lot of memory, and Message Passing Interface (MPI) and Open Multi-Processing programs supporting distributed and shared memory respectively. Every node in the cluster is interconnected with both Gigabit Ethernet and QDR InfiniBand. The name stems from Sami folklore, in which Stallo is a Sami therianthrope wizard.

Table 2.3: Aggregated hardware specifications of Stallo. Derived from [1]

Peak performance	516 TFLOPS
CPU	Intel® Xeon® Processor E5-2670 (20M Cache, 2.60 GHz) × 608 Intel® Xeon® Processor E5-2680 (20M Cache, 2.70 GHz) × 1040
Memory	26.2 TB
Internal storage	155.2 TB
Centralized storage	2 PB
Operating system	CentOS (Rocks)

Measuring expenses is different when using Stallo, as researchers are granted a given amount of CPU time, typically in the hundreds of thousands of CPU hours, to be used within a time interval.

2.4.5 ELIXIR

ELIXIR [14] is a platform consisting of several national bioinformatics institutes, as well as the international EMBL-EBI, that have joined forces to increase capabilities within life science research. Their work consists of forming

a coherent infrastructure, encompassing a collaborative effort between Europe's national and international resources in analyses and archiving of big biological heterogeneous data, coordinated by ELIXIR [15]. Their open-access infrastructure provides easy access to sustainable, community-standard data resources, and is intended to play an important role in life science projects across Europe for research purposes within medicine, bioindustries, and society. It is built on the notion of Nodes, which are national bioinformatics centres, and the EMBL-EBI, and projects from their associated life-science communities.

Chapter 3

Initial Evaluation

This chapter describes the experiments that were performed to evaluate an assessment of the cost associated to adopting AWS as the cloud service provider of a big biological data analysis service, as well as the cost of buying and maintaining a computer cluster to own. Finally, experiments were performed to establish the performance characteristics of workloads representative to our use case – mainly latencies of – interactive calculations by simulating interactive data analysis through repeatedly applying a data cleaning method to a dataset using varying parameters.

To simplify the execution of the experiments run on both AWS and `ice2`, the well-known and widely used PageRank algorithm was employed to serve as the computational element of the experiment. Although not an algorithm processing biological data, it still serves as an example of archetypical graph-based biological computations. In addition to the breadth that PageRank represents, it was also an incentive that S3 already stores the entire Common Crawl Corpus (a myriad of WARC files) as DAAS, serving as the input for identifying source websites and the websites adjacent to them. This allowed not only for unveiling performance characteristics of AWS EMR, but also the characteristics of the interaction between EMR and S3, especially by recognizing network features. The time elapsed for computations to complete will imply the cost entailed using AWS. Experiments on the locally owned cluster was performed as a contrasting environment to the AWS EMR cluster, to identify differences and similarities with regards to cost and ease of use. Lastly, data cleaning by simulating k-mer distribution optimization was applied to DNA sequencing data in order to estimate the responsiveness of a hypothetically interactive data cleaning method, which can be used by, and be useful

for, biologists. Interactive tools are latency-sensitive applications, because the user is passively waiting for a response, meaning the latency needs to be reasonably low in order to be applicable in practice.

The environments of local and remote clusters are pretty much the same, except differing hardware and software, calling for different tuning configurations in Spark. We found that estimating the amount of temporary storage needed on disk was hard. We did tests using different types of node setups on AWS and found that the largest PageRank set expands a lot. We also found that network interaction between EC2 and S3 is hurting the performance when downloading input files, and that it of course scales with amount of tasks pulling data. Moreover, we found that renting resources is significantly more expensive than purchasing a cluster to own locally, based on Norwegian prices (not including maintenance, power, and Internet costs).

The data cleaning tests showed that doing the actual computation takes too long to do it interactively, averaging almost 3 minutes per run using the locally available cluster. This is due to the data large data generation of the method, with different results, meaning the in-memory computation will only improve the data access of the original dataset.

3.1 PageRank Experiments

To showcase and reveal the performance characteristics of the different clusters used, including information on the inter-entity interaction, experiments were conducted using the PageRank algorithm, in furtherance of revealing the overhead of having input available in HDFS in contrast to downloading the input data needed across services (possibly data centers), as well as being able to see differences in run times based on the amount of memory available, and the cost it would entail if doing the PageRank computations using rented resources.

Although not biological data, extracting links from WARC-files is data-intensive and running PageRank on the extracted links is representative of big data analytics workloads, thus it will give insight in the performance characteristics of the different infrastructures on which the experiments were conducted, and later the cost it will entail when using an AWS EMR cluster. Table 3.1 lists the aggregated specs of the two clusters used in the experiments. All experiments were run using the Spark framework on top of Hadoop. The

stages of the experiment are shown in Figure 3.1, and all experiments were run with 10 iterations of PageRank.

Table 3.1: Aggregated specs of the `ice2` and AWS EMR clusters

Cluster	# threads/cores	Memory	Storage
`ice2`	80	320 GB	40,000 GB
`m3.2xlarge` × 10	80	320 GB	1,600 GB

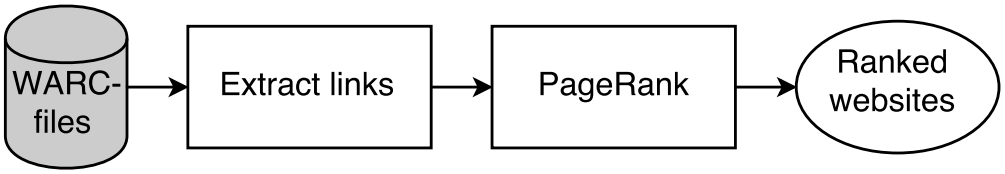


Figure 3.1: PageRank experiment stages

3.1.1 Experimental Setup

What resources that are available when running Spark on YARN is determined by the YARN configuration file. The resources are measured in number of virtual cores and amount of DRAM available, and is set up on a per-node and per-container basis. Spark programs need a driver, effectively reducing the number of nodes by one when running in client-mode and the aggregated resources available to Spark are not reflected in the resources listed in Table 3.1. The resources available to YARN on the two clusters are listed in Table 3.2. Recall that `ice2` has 10 nodes, including the front end, which is not part of the YARN configuration of the cluster, thus its starting point is 9 nodes. Note also the discrepancies in both DRAM and CPU resources between the two. The AWS EMR cluster run using Spark version 1.3.1, while the `ice2` cluster was run with version 1.3.0.

Table 3.2: Cluster metrics as seen by YARN

Cluster	Nodes	Virtual Cores	Memory
`ice2`	8	64	80 GB
EMR cluster	9	72	202.5 GB

Table 3.3 lists the configuration options used for Spark for all tests, unless otherwise stated. The persistence level translates to the cached RDDs being

serialized and are allowed to spill to disk upon exhausting the available DRAM. Two executors per node minus one was used, as the application master runs in the last available container. The Akka frame size was increased only when doing the actual PageRank, as it requires a lot of shuffling. The memory overheads refer to how much memory is allocated for things like Java Virtual Machine (JVM) overhead. The number of partitions used was calculated as number of WARC files $\times$ 12, and was repartitioned at the linkextractor stage, yielding $\{144, 1440, 14400\}$ partitions respectively for the datasets, in ascending order.

Table 3.3: Spark configuration (logical cores)

Attribute	`ice2`	EMR cluster
Persistence level	`MEM_AND_DISK_SER`	`MEM_AND_DISK_SER`
Driver memory	8 GB	23 GB
Driver cores	8	8
Akka frame size	128 MB	128 MB
AM memory	4 GB	9 GB
AM cores	4	4
AM memory overhead	1 GB	2 GB
Executor memory	4 GB	9 GB
Executor cores	4	4
Number of executors	15	17
Executor memory overhead	1 GB	2 GB

To perform the PageRank experiments, datasets readily available in S3 were used. For demonstrating the characteristics the data used needs to be big enough for the evaluations to be meaningful, and the different sizes of the datasets will show how the environment responds to more input. The experiment was run with four different input sizes, increasing the amount of files with an order of magnitude per input ($\{1.2 \times 10^1, 1.2 \times 10^2, 1.2 \times 10^3\} \approx$ {11 GB, 107 GB, 1071 GB}), all of which a subset of WARC-files from the July 2015 crawl of Common Crawl[1]. The files used are listed in the `glob`s of Listings 3.1, 3.2, and 3.3, in increasing order. The fields altered are highlighted for emphasis. Each file is approximately 900 MB in size, and because the file sizes vary and as a result of the `glob`-patterns used for S3, the values are not precise orders of magnitude apart. The input files are equal parts of the following segments:

[1] `http://blog.commoncrawl.org/2015/08/july-2015-crawl-archive-available/`

```
{
1438042981 460.12 ,
1438042981 525.10 ,
1438042981 753.21 ,
1438042981 969.11
}.
```

Listing 3.1: Input files dataset #1

```
'/1438042981[4579][0-9][0-9][.][0-9][0-9]/warc/CC-MAIN
   -20150728002301-0000 [0-2] -ip-10-236-191-2.ec2.internal.
   warc.gz'
```

Listing 3.2: Input files dataset #2

```
'/1438042981[4579][0-9][0-9][.][0-9][0-9]/warc/CC-MAIN
   -20150728002301-000 [0-2][0-9] -ip-10-236-191-2.ec2.internal.
   warc.gz'
```

Listing 3.3: Input files dataset #3

```
'/1438042981[4579][0-9][0-9][.][0-9][0-9]/warc/CC-MAIN
   -20150728002301-00 [0-2][0-9][0-9] -ip-10-236-191-2.ec2.
   internal.warc.gz'
```

Note that there is a major difference between the two clusters in the way input data is accessed. The `ice2` cluster reads input data straight from HDFS, while the AWS EMR cluster reads from buckets in S3. The latter is both due to the limited amount of storage it has in HDFS (1600 GB), but it will also reveal the efficiency of interaction between EMR and S3 and is also the more sensible choice when using the AWS platform. The exact same datasets were used on both infrastructures.

3.1.2 Results and Discussion

All experiments were only ran once, and are thus not representative of average case run times, but still gives insights to the performance characteristics. Tables 3.4 and 3.5 show the results of the two stages of the experiment, extracting links and PageRank. Figures 3.2 and 3.3 illustrate the results of the experiment in logarithmic axes graphs.

Extracting links on `ice2` looked to scale linearly for the two larger datasets, but did take a little longer for the smallest one. This is due to not saturating

the executors when having only 12 WARC files available for processing.

When extracting links using the EMR cluster, data was retrieved from S3 at an average speed of approximately 200 megabytes per second. As expected, this hurt the performance on this cluster. Additionally, speculation was enabled for this stage on the largest dataset alone, because of having severe problems with straggling tasks when downloading the last few WARC files out of 1200, which may have added some extra overhead. Speculation marks tasks that are slower than given parameters (e.g., x seconds slower than the median task completion time) and restarts the task, with the possibility of doing duplicate work.

The datasets do not scale linearly for the clusters, likely due to the last one having to store data temporarily to disk, as it does not fit in memory alone. There is a smaller gap in the relative time difference of the largest dataset, as opposed to the smaller ones. This is probably a result of the larger amount of DRAM available on the EMR cluster. However, the graphs do look to follow the same trend, with the EMR cluster gaining a bit on `ice2` on the largest dataset, as priorly stated.

Table 3.4: Runtime of extracting links (truncated to nearest second)

Cluster	12 (≈ 11 GB)	120 (≈ 107 GB)	1200 (≈ 1071 GB)
EMR × 10	221 s	761 s	6674 s
`ice2`	142 s	518 s	5409 s

`ice2` needed to spill data to disk on both of the two larger datasets, although a great amount on the middle one, when doing the PageRank. When running the large one, a large fraction of data was stored to disk, which, as can be seen, greatly affected the performance. The same is true for the EMR cluster, for the largest dataset, although with a larger fraction cached in DRAM, the impact of which can be inferred from the execution times.

We had trouble using both 10 × `r3.xlarge` (800 GB) and 10 × `m3.2xlarge` nodes (1,600 GB) because of local disk space limitations when running the largest PageRank experiment. It is hard to estimate the amount of data that is going to be spilled to disk, and how much extra storage space is needed on disk, depending on the installed software, how the disks and HDFS are set up, and so on.

Due to time and budget limitations, the last experiment doing PageRank was

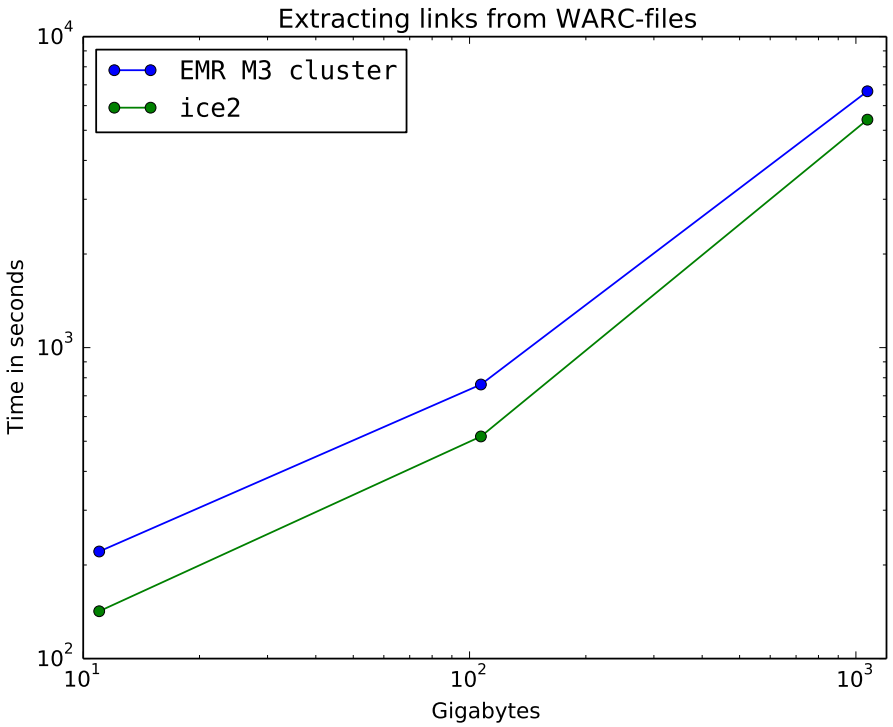


Figure 3.2: Graph showing execution time of extracting links

not successfully completed. Not only is the estimation of hardware resources needed a difficult task, but also the Spark tuning on different hardware and software is a challenge. The parameters used for Spark on the local cluster did not always work on the remote clusters, which leads to debugging while paying per hour used. With large datasets and long running times, this posed a bit of a problem.

If only taking performance into consideration, the cluster with the best hardware overall will always be the best choice. If the cluster has enough disk space to store all input data in HDFS, this is faster than having to retrieve data from a remote source, due to the overhead downloading entails. This is not a feasible solution with increasingly larger datasets, and some remote storage is needed, introducing the problem of being reliant on other services and the performance of interacting with them. It eventually boils down to the speed at which data can be retrieved from the remote source, as this will be the bottleneck. Thus, the network link between the compute and storage resources are of paramount importance, and e.g., AWS have both EC2 located and S3 replicated in Ireland, for instance, likely providing better latency and

throughput than interacting with S3 in Ireland from `ice2` in Norway. On the contrary, for `ice2`, an increase in performance over, for example interacting with S3 in Ireland, could be achieved if storage could be provided in-house for the service, e.g., at the university where `ice2` resides. Testing performance of interaction between `ice2` and S3 could be interesting, but outperform the AWS EMR and S3 combo, the storage service needs to be located geographically close to the cluster. The easiest, most reliable, and realistic approach is to go for the AWS stack, removing oneself from all hardware, replication, and maintenance related issues, and having the ability to scale up or down at our convenience.

Table 3.5: Runtime of PageRank experiments (truncated to nearest second) on the extracted links

Cluster	12 (429 MB)	120 (4.2 GB)	1200 (42.1 GB)
EMR × 10	94 s	1111 s	N/A
`ice2`	87 s	1601 s	69270 s

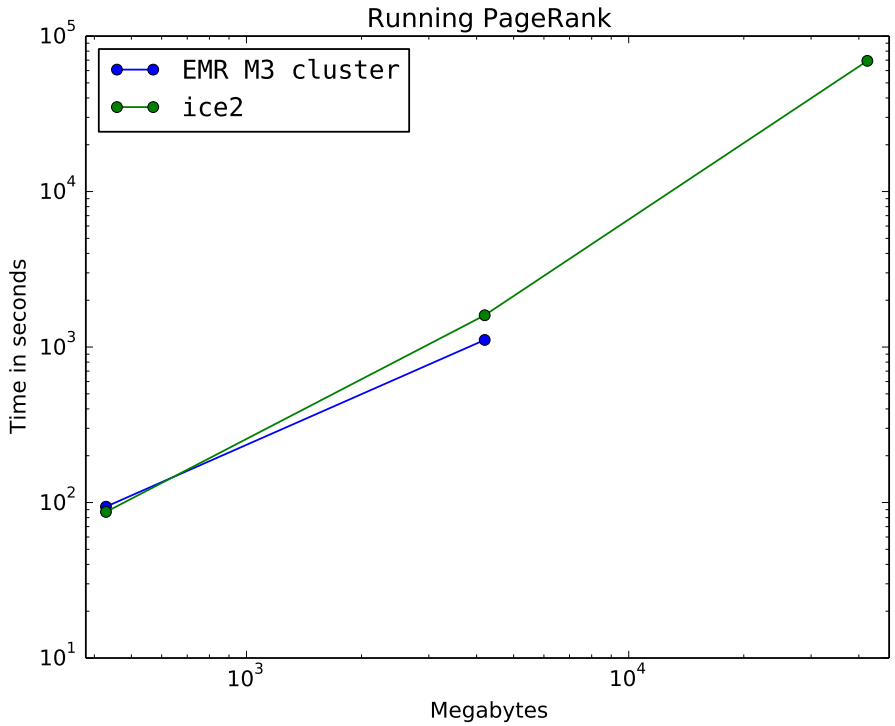


Figure 3.3: Graph showing execution time of PageRank

3.2 AWS Expense Assessment

AWS EMR offers three main pricing options, all calculated by hourly rates: on-demand instances, Spot instances, and reserved instances. On-demand instances provides, as the name suggests, on-demand service and is thus the most flexible, but also the most expensive option. Spot instances lets a client bid on idle EC2 instances, making it the cheapest option, but the nodes in a Spot cluster might be interrupted if the Spot Price exceeds the client's bid rate. This means that the bid rate of the driver of a Spark application should be reasonably larger than the current Spot Price to ensure that it does not get interrupted (e.g., extrapolated from maximum Spot Price values over the past month); worker failure may be acceptable, as it will not crash the running application. Lastly, reserving instances allows a client to reserve in advance an amount of instances for a given interval of time (1 or 3 years) at a discounted hourly rate. Note that there are no discrepancies in hardware or software resources provided across the different pricing options.

The Spot instances option gives the best return on investment, followed by reserving instances for three years, and finally the on-demand option. The hourly price per instance of `m3.2xlarge` for the different pricing options, at the time time of this writing (2015-12-08, 16:17), is listed in Table 3.6[2].

Table 3.6: Hourly rates of the AWS EMR pricing options [11, 2]. Reserved price based on 3-year term

Pricing option	Amazon EC2	Amazon EMR	Total
On-demand	$0.5850	$0.0900	$0.6750
Spot Price	$0.0809	$0.0900	$0.1709
Reserved	$0.2673	$0.0900	$0.3573

As can be deduced from the table, the Spot Price is almost one fourth of the price of on-demand, and reserved about half. Thus the optimal choice would be going for Spot instances, closely monitoring the Spot Price and comparing it to the current bid rate, as this pricing option can potentially reduce costs by a lot. The cost of the different pricing options for one hour of the cluster used is $6.75 for on-demand, $1.709 for Spot, and $3.573 for reserved instances.

Using these prices, extracting links from the largest dataset would cost $3.418 using the EMR cluster, and PageRank would cost $34.18 when exemplified

[2]All listed prices are configurations using Linux in the EU (Ireland) region.

with the `ice2` results, in lack of the result from EMR, which likely would be less due to more DRAM when doing the iterative PageRank. However, these are all numbers using the minimal cut-off price, meaning in reality the bid price would need to be a bit larger in order to get priority for node provisioning. If running a service, the driver bid price would have to be significantly larger than the current Spot Price to ensure that it stays up. The bid price of the slave nodes could be monitored over time and adjusted accordingly, as well as elastically being able to scale up or down if needed, in order to be able to deliver the service to our standards.

In addition to the EMR cluster, extra storage is needed from S3 because of the small amount of storage available in its HDFS (1,600 GB for the projected cluster). Table 3.7 lists the storage prices for the different S3 storage classes: Standard is the most expensive, followed by Infrequent Access and finally Glacier. It is also worth noting the discount based on the amount stored per month if choosing the Standard storage class. Another possibility is that it is not improbable for S3 to be interested in storing for example the data of ENA as DAAS, as they currently already store several large biological datasets; maybe even an exceptionally popular big biological data service running on AWS can prompt them to do so. Having our data readily available in S3 for free would be advantageous, and would probably prompt the cluster of our service to be deployed on AWS EMR to benefit from this.

Table 3.7: Storage prices per GB per month for S3 [3]

Storage class	$\leq$ 1 TB	[1, 50] TB	[50, 500] TB
Standard	\$0.0300	\$0.0295	\$0.0290
Infrequent Access	\$0.0125	\$0.0125	\$0.0125
Glacier	\$0.0070	\$0.0070	\$0.0070

Assuming a storage need of 100 TB, the monthly cost of each of the pricing options is \$2925.5 for Standard, \$1245 for Infrequent Access, and \$700 for glacier. Data transfer between S3 and EC2 in the same region is free [3].

Our service would probably benefit from choosing the Standard storage class, as it do not involve any retrieval fee, as opposed to Infrequent Access. This, of course, depends on the expected amount to be retrieved from S3, and it would be best to determine the amount of data that can be retrieved before it becomes detrimental in terms of cost to store data in Infrequent Access, but it is more than likely that our service would be best off with the Standard class.

3.3 ice2 Expense Assessment

The ice2 cluster consists of 10 nodes including the front end. It is built using standard Dell T3600 workstations, a gigabit switch, and a network-attached storage (NAS) box, in addition to network interface controllers (NICs), cables and racks. Buying a similar cluster today in Norwegian prices would in total cost around $35,000. The workstations would cost about $2,300 each, the gigabit switch about $2,300, and the NAS box around $7,000, in addition to the rest of the miscellaneous hardware needed.

This setup is designed for a specific environment, as it is located at a university. Access to cheap surface areas, and "free" power and Internet (provided by the university), are the main factors benefiting choosing workstations, as they provide the lowest price to compute performance ratio in this environment, as opposed to using server racks. For most real-world environments, these are factors that needs to be added to the expenses, likely ending up with a very different composition as the most cost-efficient solution.

Installation and maintenance costs are hard to predict, but an estimate of 40 hours is a reasonable estimate for setting up a cluster with basic software configurations. In Norway this would entail an expense of around $1,600 for a Senior Engineer to set the cluster up. Maintenance depends on several factors, for example what and how much needs installing and updating. It is safe to assume a couple of hours weekly for updating operating systems and applications, in addition to arbitrary disk crashes (there have been 3 disk crashes over a period of 2.5 years on ice2).

It was bought with a 5 year warranty, which is regarded as the lifespan of the cluster. If the cluster was used every hour for 5 years, a rough estimation of the hourly cost (not including installation and maintenance) can be calculated as

$$\frac{\text{Price of cluster}}{\text{Hours in five years}} = \frac{\$35,000}{(24 \times 365 \times 5) \text{ hours}} = \$0.799 \text{ per hour} \qquad (3.1)$$

This estimate is about half that of the Spot, less than 1/4 of reserved, and around 1/8 of on-demand cost of 10 m3.2xlarge instances, and the Spot Price is not anticipated to diminish over time.

Exemplifying with the time used for running the PageRank experiments on ice2, extracting links would have cost $1.598 and running PageRank would

have cost $15.98, if paying at the beginning of each hour.

3.4 Interactive Data Cleaning

As mentioned above, interactive k-mer distribution calculations were simulated in order to assess the execution time of each step of tweaking on data from a FASTQ file containing genomic DNA sequencing data, serving as the biological data cleaning method. In order to follow the terminology used, a short introduction to basic notions related to the biology and bioinformatics is provided in the ensuing section.

3.4.1 Biological Background

Metagenomics A genome contains all of an organism's genes; its complete set of DNA. It consists of around 3 billion DNA base pairs and holds all the hereditary information about that organism [31]. *Genomics* is the analysis of genomes, on an organism-level, in furtherance of better understanding the means of the organism and its evolution [45]. Metagenomics seek to understand complex communities by analyzing their genetic composition, thus capturing the dynamics of these communities in a way genomics cannot [45], which can contribute to life and earth sciences, and biomedicine, amongst others.

DNA DNA contains the genetic information of – and is found in all – organisms. It consists of the nucleobases adenine (A), cytosine (C), guanine (G), and thymine (T), which pairs up and forms base pairs. The bases are encoded in *sequences* such that the order of the bases determine the genetic information [32]. A nucleotide is a nucleobase connected to a sugar molecule and a phosphate molecule, and nucleotides in turn form the DNA in the shape of a double helix [32].

DNA sequencing Sequencing is the deciphering of an organism's genetic information by determining the correct ordering of the nucleobases. With the next-generation sequencing technologies, efficiently performing low-cost sequencing of DNA (whole genomes) helps drive several research fields, including biology and medicine [61].

FASTQ FASTQ is the de facto standard file format for storing and sharing DNA sequencing data [37]. It improves on the FASTA format by in-

cluding quality scores from the Phred quality scale to each nucleobase in all sequences. One read in the FASTQ format consists of four lines: (1) title and optional description, (2) raw sequence, (3) optional repeat of first line, (4) quality scores.

K-mer In nucleotide sequence analysis, k-mer analysis can be used for sequencing coverage estimation, repeat detection, and preparation for de novo assembly [73]. It is defined as all substrings of a DNA read sequence of length k, and is a specialization of the n-gram concept of computational linguistics. An example is given below in Listing 3.4.

Listing 3.4: Example k-mers

```
// Original string
ACAGTCA

// K-mers, k = 4
ACAG
 CAGT
  AGTC
   GTCA
```

However, most contemporary assembly algorithms represent k-mer prefixes and postfixes as de Bruijn graphs, as representing all k-mers in full does not scale to the very large [38].

3.4.2 K-mer Experiments

The value k of k-mer is the amount of nucleotides per subsequence, and hence the maximum amount of k-mers is 4^k [50], assuming 4 possible values (n^k for n possible values). This means that the data expands exponentially by a factor of k, and the memory requirements of the platform used also increases relative to the k value.

Typically, the response time of an interactive browser-based program should be associated with the complexity of the task (8 – 12 seconds for a complex task) and response times greater than 15 seconds disrupts the work [47].

The test setup was almost identical to the one described above for PageRank (§3.1.1), with increased serialization buffers and result sizes. Table 3.8 shows the arithmetic mean, median, and standard deviation of doing the k-mer computations 100 times, with varying head and tail cut-off values.

The data used was raw genomic sequencing data from a marine bacterium alcanivorax, consisting of four FASTQ files with a combined size of 2.2 GB, repartitioned by a factor of 40 (160 partitions) to increase parallelism, before doing the k-mer computation. Note that the sample execution times are measured using $k = 20$ and after an initial run-through to cache the dataset serialized in memory prior to gathering the samples used for statistics.

Table 3.8: Results of 100 k-mer experiment execution time samples

Arithmetic mean ($\bar{x}$)	Median ($\mu_{1/2}$)	Standard deviation (σ)
170.13 s	169.53 s	19.39 s

The latency is not acceptable for an interactive service, as is. This might have to do with unoptimized code, but the major issue is that the data grows extremely large when finding k-mers. The program does not benefit too much from in-memory computations, as the exponential amount of data growth needs to be computed each time the parameters are changed, yielding only a speedup by having the original dataset in memory. A better solution is probably to let a user start a program supplying a value k, and letting the program computationally analyze and find the optimal cut-off values depending on different parameters provided by the user, and not have it be interactive. It also requires a great deal of extra storage space, and is write intensive because of the intense data generation.

3.5 Future Work and Possible Optimizations

There are numerous ways of optimizing distributed computing environments across several dimensions, like the configuration of software, hardware, the software-hardware interaction, and runtime parameters used.

Regarding software, it is often times the latest version that provides the most efficient executions, as optimizations usually are added over time.

Compression algorithms must be chosen with respect to preference. Choices include the best compression to compute ratio, best compression, and speed. If storage is scarce, then compression might be more valuable than compute power; if there is an abundance of storage available, compute power might be more available than compression. Common compression algorithms in

big data include Snappy, gzip, bzip2, LZO, and LZ4. gzip compression is currently in use.

Different serialization methods should be considered, like Apache Thrift [9], Apache Avro [29], and Google Protocol Buffers [23], which produces extensible binary formats. JavaScript Object Notation (JSON) is the serialization technique currently in use.

File formats are also important, and among the options are Optimized Row Columnar (ORC) [5] files and Parquet's [6] columnar format, which should be explored further. Parquet integrates with several serialization techniques, opening for scenarios of e.g., having Avro as the in-memory representation, while using its own format in stable storage.

There is currently a lot of progress in the world of storage hardware. HDDs and SSDs are familiar hardware, and SSDs is the better choice for both performance and reliability [28]. Non-Volatile Memory (NVM) Express is an interface for Peripheral Component Interconnect (PCI) Express SSDs that increases read and write throughput by doing it in parallel. Another newer technology is the 3D XPoint^{TM} non-volatile memory, which is a collaborative effort of Micron and Intel, claiming to have exponentially greater durability and significantly lower latency than NAND [20]. It will be interesting to see what impact these new storage systems have on the big data ecosystem, if any.

Spark can be run as standalone or it can be run using cluster managers. The cluster managers are YARN [66] and Mesos [46]. It could be an interesting comparison to see a performance comparison of using the different resource managers.

Finally, there are a large number of parameters that can be changed and tuned for performance in Spark, ranging from how many executors that should be run, amount of memory and virtual cores per executor, different memory overheads for the JVM, and much more. There are an almost endless amount of combinations using different attributes in efforts to optimize programs.

Regarding future work, as mentioned, the main goal is to provide a general service, providing the frameworks and architecture for doing big biological data analyses with the client providing scripts that describe an algorithm and what data it should be run using, and the service should take care of the rest.

Chapter 4

Background

This chapter contains information regarding different concepts, methods and software relevant to the project. It explains the key concepts of several big data frameworks that were under consideration and used, and a brief description of a genomic analysis pipeline developed at the New York Genome Center, a system built using Spark, Avro, and Parquet, and short explanations of a couple of bioinformatics algorithms from the Raphael Lab.

4.1 Big Data Processing Frameworks

To handle big data, tools specialized for the task is needed. The following sections explore different types of big data frameworks developed at Google, for Apache, and at Microsoft.

4.1.1 Google

Google is a large American technology company providing several Internet services, their most famous service probably being their search engine Google Web Search. They are one of the pioneers on the topic of big data handling, and as such has developed many frameworks to handle their all their data. Some of which are described in the following.

MapReduce

MapReduce is a programming model first described by Dean et al. [41] at Google. It is a fault-tolerant parallel processing system designed for data-intensive computation, and is intended to be run on a cluster of commodity machines, exploiting data locality of having both the data and compute capacity on a given node through scheduling. Programs are written using map and reduce functions that are transparently parallelized and deployed by the runtime system, also claiming that MapReduce is an easy abstraction to work with. The map function aggregates key/value pairs of some input key/value pairs by locally grouping the values by keys. The reduce function takes the intermediate results of the map and groups the values by keys as well in an attempt to reduce even further. Hadoop MapReduce is an open-source implementation of this programming model.

The canonical example illustrating MapReduce is word count, in which a string is tokenized by words and assigned a value, serving as the key/value pair (*word*, 1). The set is then grouped by key and the values combined, resulting in a list of words and their associated occurrences in the string (*word*, *occurrences*).

Google File System

The Google File System [43] is a file system developed to run on a clusters of commodity machines and is designed to support data-intensive workloads in a distributed environment. It is scalable, reliable, highly available, provides accumulative performance and employs relaxed consistency. Identifying that failures are commonplace in large distributed systems emphasizes the need for fault tolerance and recovery, which is handled by constant monitoring. The system is also optimized for larger file sizes (multi-GB) by introducing the concept of chunks, as well as for appending and sequential reading, recognizing the different than traditional access patterns. It was in extensive use at Google shortly after its inception, facilitating the generation and processing of data for both services and research efforts. HDFS is an open-source Java implementation of the Google File System.

Bigtable

Bigtable [35] is a storage system developed at Google for distributed storing of structured data. Being distributed, it provides high availability and performance, and scales to triple-digit size clusters storing petabytes of data. It shares some of the same strategies as databases, but does not support a relational model, rather allowing clients to specify and have dynamic control over layout and format as well as letting clients decide whether to serve data from memory or disk. Data is represented as tables which are sparse sorted maps containing rows, columns, and timestamps. Bigtable was, at the time, in use by several Google projects and services, demonstrating its wide applicability by supporting widely varying needs with regards to latency and size.

Spanner

Spanner [39] is a temporal multi-version database. It is scalable, fault tolerant, and globally distributed by automatic sharding and migration of data across data-centers using Paxos [48] state machines. Resharding is done in response to failures and for load-balancing purposes, and clients may specify data location, replication factor and placement to what best fits the needs of their application. Spanner provides external consistency and general-purpose transactions at high performance accessible through a SQL-based query language. Data is versioned and timestamped at commit time and organized in semi-relational tables. Applications can read data at old timestamps. The TrueTime API keeps clock uncertainty small by using Global Positioning System (GPS) and atomic clocks; if the uncertainty grows too large, Spanner slows down. TrueTime is instrumental in facilitating the serialization order on timestamps which accomplishes external consistency.

Pregel

Pregel [52] is a computational model designed to handle large scale graphs of up to billions of vertices and trillions of edges using clusters of commodity machines in a fault tolerant manner. The distributed platform exposes an API that is flexible enough to enable the expression of a wide range of graph algorithms. Programs are expressed as a sequence of iterations in which each vertex is manipulated on in parallel by a user-defined function,

communicating by receiving messages from a previous iteration and sending to a succeeding iteration. The computations are done in DRAM, providing fast response times over iterations of the same data by acting as a cache. PageRank, shortest path, bipartite matching, and semi-clustering are among the tested and proven algorithms expressed using the Pregel platform.

4.1.2 Apache Software Foundation

The Apache Software Foundation (ASF) supports a great number of open source projects by providing financial and legal support [18]. It is a non-profit corporation encouraging collaborative development of the projects. Their best known projects are likely the HTTP Server and OpenOffice.

Hadoop YARN

YARN [66] is a resource manager that applications in Hadoop can run on top of and rely on for scheduling and handling resources. Its architecture consists of a per cluster Resource Manager that allocates containers to applications to be run on given nodes in the cluster in a dynamic fashion, in collaboration with Node Managers running on the worker nodes responsible for the inhabited node's resources. A container is an abstraction that comprises the delegated logical resources. The Application Master is responsible for coordinating the execution plan by requesting resources from the Resource Manager and do the execution of a program in a fault tolerant manner, and is itself run as a container in the cluster. YARN is scalable, efficient and enables large numbers of frameworks to share and simultaneously use a cluster.

Spark

Spark [8, 71] is a cluster computing framework written in Scala, originally built on top of the Mesos [46] platform. It was developed at the AMPLab (Algorithms, Machines, and People) [4] of UC Berkeley and serves as the processing engine of their Berkeley Data Analysis Stack (BDAS). The BDAS' raison d'être is making sense of big data, and consists of several self-built and third-party components. Spark is designed to support applications unable to be efficiently expressed as acyclic data flows such as graph processing and machine learning, in a scalable and fault tolerant manner similar to

MapReduce [41]. It does so by introducing two abstractions: RDDs [70] and parallel operations on these datasets. These abstractions allows for efficiently expressing several existing programming models, including, but not limited to, MapReduce and Iterative MapReduce, SQL, and Pregel [52].

RDDs are read-only (immutable), partitioned collections of objects created through operations on data in stable storage or other RDDs. This parallel data structure enables data reuse by persisting intermediate results in memory, improving the performance of several types of applications, most notably iterative algorithms and interactive data mining tools [70]. The RDDs expose an interface of *transformations* that executes one operation on many data elements. Transformations define new RDDs and are lazily computed to support pipelining. Efficient fault tolerance is achieved by logging a dataset's lineage instead of the data itself (the lineage consists of all transformations used to build a certain dataset) allowing for recomputation of certain partitions whenever needed, without the overhead of replication. Upon failure, only the lost partitions are recomputed, which may be done in parallel. After creation, an RDD may be manipulated using operations that return a value to the driver program or to write data to a storage system, referred to as *actions*. An RDD is represented as a Scala object in Spark, statically typed and parametrized by an element type.

In Spark terminology, a developer writes a driver program that connects to a cluster of workers. At runtime, a driver program containing the control flow of an application launches multiple workers that read data from some distributed file system and may persist to memory the computed RDD partitions. The Spark scheduler builds a directed acyclic graph (DAG) consisting of execution stages based on the lineage graph of the RDD upon which the action is performed [70]. Tasks may also be scheduled based on data locality using delay scheduling, and in case of memory exhaustion, data is spilled to disk and the performance of the RDDs gracefully degrade [70]. Spark was designed to enhance the Hadoop stack and thus supports HDFS, HBase, SequenceFiles, as well as S3, amongst others.

RDDs can be persisted in memory either as serialized data or as deserialized Java objects, and it can be persisted in non-volatile storage. Furthermore memory is managed using the least recently used (LRU) eviction policy on RDDs currently residing in memory. By storing Java objects in memory, the cost of deserialization and I/O can be circumvented, making Spark perform better than Hadoop MapReduce in graph and iterative machine learning applications [70].

Table 4.1: Examples of transformations and actions in Spark. Derived from [70]

Examples of transformations:

•	*map*	Each item is passed through a provided function
	(f: T ⇒ U):	`RDD[T]` ⇒ `RDD[U]`
•	*flatMap*	Like *map*, but can map each item to multiple output items
	(f: T ⇒ Seq[U]):	`RDD[T]` ⇒ `RDD[U]`
•	*filter*	Filter elements of the dataset based on a provided function
	(f: T ⇒ Bool):	`RDD[T]` ⇒ `RDD[T]`

Examples of actions:

•	*count*	Count elements in dataset
	():	`RDD[T]` ⇒ `Long`
•	*reduce*	Aggregate elements of dataset using a provided function
	(f: (T,T) ⇒ T):	`RDD[T]` ⇒ `T`
•	*save**	Save RDD output to storage system
	(*path*):	`RDD[T]` ⇒ `storage(`*path*`)`

Spark also includes access to, and interfaces for, GraphX, MLlib, and Spark

Streaming, to mention some:

GraphX [68] is an open-source graph processing framework that provides a way of efficiently expressing graph computation using Spark, and it does so by exploiting the benefits of both data-parallel and graph-parallel systems. Graphs are represented as tabular data, and the main abstraction of GraphX is that of the resilient distributed graph (RDG), which is an extension of the RDD abstraction, providing a set of computational primitives that also minimizes the data movement during computation. The abstraction is claimed to simplify graph computation, transformation, and construction, and has been tested and proven by being used to implement the Pregel and PowerGraph APIs.

MLlib [56] is the largest open-source distributed machine learning library for Spark. It exploits the fact that Spark is great for iterative computations, due to in-memory computing, and that many large-scale machine learning algorithms are inherently iterative. Among the features supported are linear models, naive Bayes, alternating least squares, and k-means clustering and optimizations on these. A pipeline API is also provided by MLlib, which supports multi-stage machine learning pipelines by simplifying development, tuning, and the ability to swap out algorithms in different stages.

Spark Streaming [72] provides a stream programming model called discretized streams (D-Streams) as an extension to the Spark framework. The motivation behind streaming is that data often times is received in real time, and the freshness of data processed is an important aspect. D-Streams do a sequence of deterministic computations on data delimited by small intervals of time, provides strong consistency, and efficient fault tolerance. It supports two operator types: transformation and output operators. Transformation operators are used to create new D-Streams from some parent stream and output operators are used to write data back to some storage system.

Storm

Apache Storm [65] is a real time stream data processing system built at Twitter, that is scalable, fault-tolerant, and distributed. It processes streams of tuples that are part of a directed graph of operators (a topology) and runs on a cluster, usually on top of a cluster manager abstraction (e.g.,

Mesos [46]) and supports different partitioning strategies including shuffle, fields, and local. Storm is part of the infrastructure at Twitter, aiding in real time data-decisions made, and is designed to efficiently stabilize performance after failure scenarios.

4.1.3 Microsoft

Microsoft is a large American software company best known for developing the Windows operating systems and the Office package. They also provide several cloud and other Internet services, and have in recent times started open sourcing select frameworks.

Prajna

Prajna [49] is Microsoft's new distributed functional programming platform, which is heavily influenced by Spark in that it also seeks to utilize DRAM to its full potential by aggressively doing in-memory computing. It is an open-source platform running on .NET and F#, supporting the development of cloud services and interactive data analysis. Their contributions over Spark lies in harmonizing cloud services and data analytics using in-memory processing with their abstraction Distributed data Sets (DSets), as well as claiming to provide improved programmability, debugging, and building of systems by leveraging functional programming concepts.

DryadLINQ

DryadLINQ [69] is a programming model that includes language extensions that builds efficient distributed programs for clusters of commodity computers through transparently transforming a general-purpose language program to a distributed execution plan using a compiler. Both Dryad and LINQ are specialized for stream processing. Dryad facilitates an execution model that is flexible; LINQ (Language INtegrated Query) is used for programming with datasets using .NET and makes debugging easier as a result of its strong static typing. The compiler produces a distributed execution plan for Dryad by transforming data-parallel portions of the input program. Dryad then executes the plan in a reliable and efficient way.

4.2 Scala

Spark is written in Scala [25], which is also the language chosen to interface with Spark in this project. It is a statically typed programming language which unifies functional and object-oriented programming. It offers support for component abstraction, composition, and decomposition, and has a rich syntax and type system [57] as well as built-in mechanisms for type inference. Scala programs can interact seamlessly with Java programs and it compiles to Java bytecode which runs on the JVM, easing the adoption of Scala for both users of Java, and systems already implemented in Java.

Scala is in extensive use by several large companies, including LinkedIn, Twitter, Xerox, The Guardian, and FourSquare to mention a few [26].

4.3 A Genomic Data Analysis Pipeline

An example of a big biological data deep analysis pipeline in genomics research is described in the paper "Building Highly-Optimized, Low-Latency Pipelines for Genomic Data Analysis" by Diao et al. [42]. The motivation behind their research was to optimize genomic pipelines in order to facilitate deep analysis in shorter time, to address the problem of the ever-increasing growth of genomic data and possibly make it applicable to clinical settings. In order to extract biological meaning from the sequencing data, it needs to be worked and transformed through several different steps. They recognized that the current pipelines were I/O inefficient, that each step increases size, and the complexity of the algorithms used impedes the overlapping of pipeline stages.

The following list is a broad outline of the steps in their analysis pipeline:

(0. Preprocessing if needed)

1. Alignment:

   Short reads are aligned against reference genome using the Burrows-Wheeler Aligner, then aligned data is encoded to SAM/BAM format.

2. Data cleaning:

   Noisy data is removed.

3. Variant calling:

   Detect variants against reference genome, including small variant calls such as SNP and INDELs, and large structural variants like copy number variants, inversion, and translocation. Outputs VCF.

4. Deep analysis:

   High-level biological information of value can be produced by performing statistical analysis over a population, such as associations, causal relationships, and functional pathways.

4.4 ADAM

ADAM [54] is a framework supporting several of the common processing stages of a genomic analysis pipeline, introducing new formats, APIs, and is an open source project developed as a collaborative effort between UC Berkeley, MIT, and Harvard. They identified the current de facto formats as ineffective for distributed environments, arguing that they were designed with sequential programs in mind. Thus, they are trying to introduce new formats and, a framework using these, that are designed for distributed and parallel computing at cloud scale. It is shown to scale, and to run more efficiently than conventional genomic analysis tools in use. Their current implementation is built using Avro and Parquet for more efficient lossless compression and to allow access by database systems, with Spark as the processing engine for fast in-memory computations. The main disadvantage of this solution is that current implementations of algorithms and tools need to be rewritten to adhere to both their framework and schemata.

4.5 THetA2

THetA2 [60] is an extension of the THetA [58] algorithm, developed by the Raphael Lab of Brown University. THetA is short for Tumor Heterogeneity Analysis, and is an algorithm that process high-throuhgput sequencing data of DNA, of whole genomes and exomes, to infer tumor purity and amount of tumor subpopulations. This is motivated by the fact that most tumor samples contains a mixture of cells, both normal cells and cancerous cells, and purity is important for analyzing somatic aberrations of the tumor cells.

THetA2 is faster and can, from tumor samples, identify subclonal populations. THetA2 finds the composition of these subpopulation(s) by distinguishing the copy number aberrations, and finds the percentage of normal cells, and has been tested using both high and low coverage whole genome sequencing data, and whole exome sequencing data. It is mainly written in Python, with tools implemented in both Java and MATLAB, and is open sourced [24].

4.6 PREGO

PREGO [59], short for Paired-end Reconstruction of Genome Organization, is another algorithm developed at Brown University, by the Raphael Lab. Using paired-end DNA sequencing data, it can identify rearrangements of biological relevance in cancer genomes and reconstruct structural variants, in an efficient manner, from the reference genome. PREGO can infer the organization of a cancer genome by using adjacency and copy number information, by using existing genome assembly techniques, copy number variant prediction, and genome reconstruction and rearrangement analysis. It has been tested and shown to identify rearrangements that are consistent with established methods, for rearrangements both reciprocal and non-reciprocal using five ovarian cancer genomes. PREGO is written in Java, and its implementation is available on the Raphael Lab's [24] websites.

Chapter 5

Conclusion

We have presented a requirements analysis, initial evaluation, and background research of a big biological data processing service.

Because most biological data analyses are currently being conducted on smaller datasets, the need for a big biological data service is becoming apparent. The potential knowledge to be gained from these types of analyses may prove invaluable for several branches within the biological fields, which was the main motivation behind this work. A main goal for a future service is to make it widely usable, without having to be an expert in big data tools and management, to cater to a broader audience.

We did a requirements analysis for a given operational scenario and proposed an architecture for data management between different services. Moreover, several platforms were considered and researched in order to find the big data tools that best fit our needs. The considered platforms and frameworks were described and we reason about our decision of going for Spark on top of Hadoop YARN, and the then natural choice of going for HDFS. Infrastructures were researched in furtherance of investigating different environments for running our frameworks of choice and handling the big biological data, which were also used for running experiments, excluding the HPC cluster.

An initial evaluation was conducted to gain insight in the different cluster environments, the interaction between entities used, and to assess the cost of using both locally owned and rented, from a cloud service provider, compute and storage resources. Two different experiments were done, one to evaluate the synergy of compute and storage components in the cloud, and to get insight in the different environments. We found that tuning Spark is time-

consuming, and approximating the required storage resources can be hard. Furthermore, renting resources from AWS was found to be more expensive than purchasing a cluster that is owned, and doing data cleaning using k-mer distribution interactively results in too high latencies for it to be adapted for actual use, giving an incentive to automate the optimization.

Combined, the lessons learned during this project provide a good foundation for designing our envisioned service. In particular, we intend to use Spark and Hadoop to facilitate the processing and data management of big biological data, as these are widely used frameworks gaining increasingly more attention, with rich functionality and configuration options. Especially in Spark, almost any parameter may be tuned and tweaked until a satisfactory configuration is achieved, as well as excellent integration with the Hadoop system, making it a great fit for our service. It is important to configure Spark to utilize all of the resources available on a cluster, but also to optimize the parallelism and to balance between garbage collection churn and not having YARN kill executors. Moreover, using AWS for both the compute and storage needs for our service is compelling, as it provides elasticity, reliability, and availability, not to mention simplicity in terms of being able to ignore hardware, replication, and maintenance related issues, as well as being able to elastically add or remove resources as needed. These characteristics are favorable in a large scale distributed system, and the flexibility allows for easily expanding the service, for instance as a response to increasing popularity. Having both compute and storage resources located in the same city, e.g., Dublin, Ireland, is favorable in terms of both latency and throughput, and is another incentive to choose AWS solutions. It will be more expensive for compute resources than buying a cluster to own, but it is probably still both the most realistic and achievable solution. Data representation in terms of serialization and file formats used are of great importance, and in particular Avro in combination with Parquet is an approach that we will give some attention, inspired by the work done in ADAM [54], with increased compression ratio and facilitation of SQL queries on stored data.

References

[1] About Stallo – HPC documentation. http://hpc-uit.readthedocs.org/en/latest/general/stallo.html. [Online; accessed 22-November-2015].

[2] Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/. [Online; accessed 12-October-2015].

[3] Amazon S3 Pricing | AWS Cloud Storage Pricing. https://aws.amazon.com/s3/pricing/. [Online; accessed 20-October-2015].

[4] AMPLab. https://amplab.cs.berkeley.edu/.

[5] Apache ORC • High-Performance Columnar Storage for Hadoop. https://orc.apache.org/.

[6] Apache Parquet. http://parquet.apache.org/.

[7] Apache Spark. https://www.cloudera.com/content/www/en-us/products/apache-hadoop/apache-spark.html. [Online; accessed 21-November-2015].

[8] Apache Spark. http://spark.apache.org/.

[9] Apache Thrift - Home. https://thrift.apache.org/.

[10] AWS | Amazon EC2 | Instance Types. http://aws.amazon.com/ec2/instance-types/. [Online; accessed 12-October-2015].

[11] AWS | Amazon Elastic MapReduce (EMR) | Pricing. https://aws.amazon.com/elasticmapreduce/pricing/. [Online; accessed 12-October-2015].

[12] AWS | Amazon S3 | Storage Classes. https://aws.amazon.com/s3/storage-classes/. [Online; accessed 20-October-2015].

[13] Center for Bioinformatics (SfB) – UiT. https://en.uit.no/forskning/forskningsgrupper/gruppe?p_document_id=347053.

[14] ELIXIR Data for life. https://www.elixir-europe.org/.

[15] ELIXIR Scientific Programme 2014-2018. http://www.elixir-europe.org/system/files/elixir_scientific_programme_1.pdf. [Online; accessed 22-November-2015].

[16] EMBL European Bioinformatics Institute. http://www.ebi.ac.uk/.

[17] European Nucleotide Archive < EMBL-EBI. http://www.ebi.ac.uk/ena.

[18] Foundation Project. http://www.apache.org/foundation/. [Online; accessed 25-November-2015].

[19] GitHub apache/spark. https://github.com/apache/spark/. [Online; accessed 19-November-2015].

[20] Micron Technology, Inc. - 3D XPoint Technology. http://www.micron.com/about/innovations/3d-xpoint-technology. [Online; accessed 26-November-2015].

[21] New – Apache Spark on Amazon EMR | AWS Official Blog. https://aws.amazon.com/blogs/aws/new-apache-spark-on-amazon-emr/. [Online; accessed 21-November-2015].

[22] NorStore research data archive. https://archive.norstore.no/.

[23] Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/.

[24] Raphael Lab // Software. http://compbio.cs.brown.edu/software/.

[25] Scala. http://www.scala-lang.org/.

[26] Scala in the Enterprise | The Scala Programming Language. http://www.scala-lang.org/old/node/1658. [Online; accessed 25-September-2015].

[27] Stallo | Sigma2. https://www.sigma2.no/content/stallo. [Online; accessed 22-November-2015].

[28] Validating the Reliability of Intel®Solid-State Drives. http://www.intel.de/content/dam/doc/technology-brief/intel-it-validating-reliability-of-intel-solid-state-drives-brief.pdf. [Online; accessed 26-November-2015].

[29] Welcome to Apache Avro! https://avro.apache.org/.

[30] Welcome to Apache™ Hadoop®! https://hadoop.apache.org/.

[31] What is a genome? - Genetics Home Reference. http://ghr.nlm.nih.gov/handbook/hgp/genome. [Online; accessed 14-October-2015].

[32] What is DNA? - Genetics Home Reference. http://ghr.nlm.nih.gov/handbook/basics/dna. [Online; accessed 14-October-2015].

[33] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[34] Brian Beach. Virtual private cloud. In *Pro Powershell for Amazon Web Services*, pages 67–88. Springer, 2014.

[35] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[36] Amazon Elastic Compute Cloud. Amazon web services. *Retrieved November*, 9:2011, 2011.

[37] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[38] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[39] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev,

Christopher Heiser, Peter Hochschild, et al. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[40] Lin Dai, Xin Gao, Yan Guo, Jingfa Xiao, Zhang Zhang, et al. Bioinformatics clouds for big data manipulation. *Biology direct*, 7(1):43, 2012.

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

[42] Yanlei Diao, Abhishek Roy, and Toby Bloom. Building highly-optimized, low-latency pipelines for genomic data analysis. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'15)*, 2015.

[43] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[44] Casey S Greene, Jie Tan, Matthew Ung, Jason H Moore, and Chao Cheng. Big data bioinformatics. *Journal of cellular physiology*, 229(12):1896–1900, 2014.

[45] J Handelsman, J Tiedje, L Alvarez-Cohen, M Ashburner, IKO Cann, EF Delong, W Ford Doolittle, CM Fraser-Liggett, A Godzik, JI Gordon, et al. The new science of metagenomics: Revealing the secrets of our microbial planet. *Nat Res Council Report*, 13, 2007.

[46] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[47] John A Hoxmeier and Chris DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000.

[48] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[49] Jin Li, Sanjeev Mehrotra, and Weirong Zhu. Prajna: Cloud service and interactive big data analytics, 2015.

[50] Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects. *arXiv preprint arXiv:1308.2012*, 2013.

[51] Nicholas M Luscombe, Dov Greenbaum, Mark Gerstein, et al. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(4):346–358, 2001.

[52] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[53] Vivien Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

[54] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. Adam: Genomics formats and processing patterns for cloud scale computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207*, 2013.

[55] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[56] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807*, 2015.

[57] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[58] Layla Oesper, Ahmad Mahmoody, and Benjamin J Raphael. Theta: inferring intra-tumor heterogeneity from high-throughput dna sequencing data. *Genome Biol*, 14(7):R80, 2013.

[59] Layla Oesper, Anna Ritz, Sarah J Aerni, Ryan Drebin, and Benjamin J Raphael. Reconstructing cancer genomes from paired-end sequencing data. *BMC bioinformatics*, 13(Suppl 6):S10, 2012.

[60] Layla Oesper, Gryte Satas, and Benjamin J Raphael. Quantifying tumor heterogeneity in whole-genome and whole-exome sequencing data. *Bioinformatics*, 30(24):3532–3540, 2014.

[61] Erik Pettersson, Joakim Lundeberg, and Afshin Ahmadian. Generations of sequencing technologies. *Genomics*, 93(2):105–111, 2009.

[62] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health Information Science and Systems*, 2(1):3, 2014.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[64] Suzanne S Sindi, Selim Onal, Luke C Peng, Hsin-Ta Wu, and Benjamin J Raphael. An integrative probabilistic model for identification of structural variation in sequencing data. *Genome Biol*, 13(3):R22, 2012.

[65] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[66] 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.

[67] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. Graysort on apache spark by databricks, 2014.

[68] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[69] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[70] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[71] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[72] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[73] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. 2014.

Appendix A

Source Code

The source code can be found in the attached CD-ROM.