UNIVERSIDADE FEDERAL FLUMINENSE

ALAN LIRA NUNES

OPTIMIZING COMPUTATIONAL COSTS FOR MAPREDUCE-LIKE SPARK APPLICATIONS ON THE CLOUD

NITERÓI

2022

ALAN LIRA NUNES

OPTIMIZING COMPUTATIONAL COSTS FOR MAPREDUCE-LIKE SPARK APPLICATIONS ON THE CLOUD

Master's dissertation presented to the Graduate Program in Computing at Universidade Federal Fluminense as a partial requirement for obtaining the Master's Degree in Computing. Concentration area: Computer Science.

Advisor:
LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

Co-advisor:
MARIA CRISTINA SILVA BOERES

NITERÓI

2022

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

N972o Nunes, Alan Lira
Optimizing Computational Costs for MapReduce-Like Spark Applications on the Cloud / Alan Lira Nunes ; Lúcia Maria de Assumpção Drummond, orientadora ; Maria Cristina Silva Boeres, coorientadora. Niterói, 2022.
119 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2022.

DOI: http://dx.doi.org/10.22409/PGC.2022.m.15437839707

1. Computação em nuvem. 2. Otimização (Computação). 3. Tolerância a falha (Computação). 4. Computação paralela e distribuída. 5. Produção intelectual. I. Drummond, Lúcia Maria de Assumpção, orientadora. II. Boeres, Maria Cristina Silva, coorientadora. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

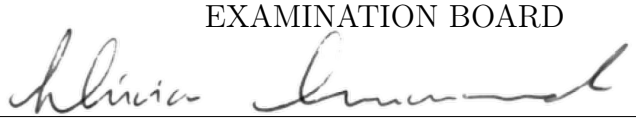
ALAN LIRA NUNES

OPTIMIZING COMPUTATIONAL COSTS FOR MAPREDUCE-LIKE SPARK APPLICATIONS ON THE CLOUD

Master's dissertation presented to the Graduate Program in Computing at Universidade Federal Fluminense as a partial requirement for obtaining the Master's Degree in Computing. Concentration area: Computer Science.

Approved in July 2022.

EXAMINATION BOARD



Prof. D.Sc. Lúcia Maria de Assumpção Drummond – Advisor, UFF

Prof. D.Sc. Maria Cristina Silva Boeres – Co-advisor, UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira, UFF

Prof. D.Sc. Claude Martin Tadonki, MINES ParisTech-PSL / CRI

Niterói

2022

To the Incarnate Word, Jesus Christ,

Dear of the Eternal Father,
Blessed of the Lord,
Author of Life,
King of Glory,
World Savior,
Desire of the nations,
Desire of the Eternal Hills,
Heavenly Bread,
Universal Judge,
Mediator between God and men,
Master of Virtue,
Lamb without blemish,
Man of pain,
Eternal Priest,
Victim of Love,
Source of Blessings,
Good Shepherd,
Lover of souls,

dedicates this work Alan, sinner.

Acknowledgment

«Sine me nihil potestis facere.» — Without me you can do nothing. (Jo XV, 5)

And though I have been unfaithful and negligent, I cannot be ungrateful and fail to acknowledge Your Goodness and extol Your Magnificence, beloved God! I adore You from the abyss of my nothingness. I give You thanks for all the benefits You have done me. What could You find in me to love me so excessively, even when my heart, stained with a thousand faults, had for You nothing but indifference and hardness? O Jesus, may songs of praise be sung to Your Most Generous Heart, which has triumphed over death and hell, deserving, in truth, all the Glory. Give me strength and courage so that, after having fought and won here on earth, I may have the happiness of triumphing with You in Heaven.

Dear professors Lúcia Drummond and Cristina Boeres, I thank you immensely for all the guidance, patience, incentives, corrections, and learning. I hope I accomplished at least half of what you planned for me, and for everything I have failed to fulfill due to my limitations, I sincerely apologize. I'd like to express my gratitude for the journey so far, and I can't wait for the incoming adventures and challenges.

I thank professors Daniel de Oliveira, Alba Melo, and Claude Tadonki for the opportunity to work together. Your ideas and suggestions were of great value for this work.

I thank my beloved father Renato and my beloved mother Neide for all their love, incentive, and understanding. I recognize how much you fought and still fight for me.

I thank my beloved brother Renan for all the incentives and for always being ready to help me.

I thank my beloved girlfriend Thamyres for all the love, patience, and affection.

I thank the C+HPC Lab research team at UFF for the opportunities and resources.

I thank CAPES for granting a scholarship for this work to be developed.

I thank CNPq and AWS for the BioCloud Project, which made this work possible.

Ce qui Te reste, ce qu'on refuse,
donnez-moi-ž'en partage, combats et courage, ô mon Dieu.

Ce qu'on rejette, ce qu'on ñ'accepté,
donnez-moi-ž'en partage, Ta croix et courage d'la porter.

Que je sois sûr de vivre à souffrir,
pour défendre la Foi, pour Ton amour mourir, ô mon Dieu.

Que je sois sûr de vivre en danger,
d'embrasser Ta croix, et dans Ta paix mourir, ô mon Dieu.

Non pas douceur, non les honneurs,
donnez-moi-ž'en partage, amers outrages des menteurs.

Non pas succès, non être aimé,
donnez-moi-ž'en partage, la haine et la rage des mauvais.

À moi, mon Dieu, la croix du mépris,
que je reste tout seul dans le plus grand oubli, ô mon Dieu.

À moi Seigneur, la gloire d'être haï,
d'embrasser Ta croix, et dans Ta paix mourir, ô mon Dieu.

Moi je ne veux ni paix ni l'or,
donnez-moi-ž'en partage, la guerre et l'orage, ô mon Dieu.

Je te demande d'un jet Ta croix,
car je n'ai le courage de deux fois la demander.

Que je sois sûr qu'Elle soit mon trésor,
d'embrasser Ta croix, avec ardent amour, sans retour.

Que je sois sûr d'avoir le plus dur,
d'embrasser Ta croix, et dans Ta paix mourir, ô mon Dieu.

(FEDELI, Orlando. ***Prière du Partage****. São Paulo, 11/19/1995 A.D.)*

Resumo

Computação em nuvem é atualmente uma das principais opções no cenário de infraestrutura computacional. Além de vantagens como o modelo de fatura *pay-per-use* e elasticidade de recursos, há vantagens técnicas quanto à heterogeneidade e configuração em larga escala. No modelo *Infrastructure as a Service* (IaaS), uma gama de recursos físicos e virtuais está disponível para alocação dinâmica, de acordo com a demanda do cliente, muitas vezes parecendo ilimitada em termos de tempo ou quantidade. Além disso, como alternativa ao modelo padrão de precificação de máquinas virtuais (VMs), os provedores de nuvem oferecem preços com desconto para o aluguel de VMs preemptivas, que podem ser revogadas a qualquer momento pelo provedor. Assim, aplicações tolerantes a falhas podem se beneficiar deste mercado preemptivo visando a redução de custos monetários. Ao lado da necessidade clássica de desempenho (*e.g.*, tempo, espaço, e energia), há um interesse no custo financeiro que pode vir de restrições orçamentárias. Com base nas considerações de escalabilidade e no modelo de preços das nuvens públicas tradicionais, uma saída esperada para a estratégia de otimização poderia ser a configuração de VMs mais adequada para executar uma carga de trabalho específica. Neste trabalho, é desenvolvida uma aplicação *Spark* baseada no modelo *MapReduce*, denominada *Diff Sequences Spark*, que realiza comparações de sequências biológicas e identifica as ocorrências de caracteres de nucleotídeos não correspondentes. Tal aplicação é executada considerando comparações de sequências de coronavírus SARS-CoV-2, que é o vírus responsável pela doença COVID-19, usando o serviço de nuvem AWS EC2 da Amazon em instâncias de VM *on-demand* (padrão) e *spot* (preemptiva). Sob a perspectiva de otimizações de tempo de execução e custo monetário, é proposta uma adaptação de um modelo de custo de execução extraído da literatura, cuja avaliação experimental obteve baixas taxas de erro. Os resultados experimentais usando tais otimizações superaram os cenários onde um usuário de nuvem inexperiente selecionaria VMs sem qualquer critério razoável. Por fim, foram alcançados reduções de custos monetários ao usar instâncias *spot* em comparação com suas respectivas opções *on-demand*, mesmo em cenários com várias revogações de *spot Workers* em um *cluster Spark*.

Palavras-chave: Apache Spark; MapReduce; Computação em nuvem; Otimização.

Abstract

Cloud computing is currently one of the prime choices in the computing infrastructure landscape. In addition to advantages such as the pay-per-use bill model and resource elasticity, there are technical benefits regarding heterogeneity and large-scale configuration. In the Infrastructure as a Service (IaaS) model, a range of physical and virtual resources is available for dynamic allocation, according to the customer demand, often appearing limitless in terms of time or quantity. Besides, as an alternative to the standard virtual machines (VMs) pricing model, the cloud providers offer discounted prices for the rental of preemptive VMs, which can be revoked anytime by the provider. Therefore, fault-tolerant applications may benefit from this preemptive market seeking monetary cost reductions. Alongside the classical need for performance (*e.g.*, time, space, and energy), there is an interest in the financial cost that might come from budget constraints. Based on scalability considerations and the pricing model of traditional public clouds, an expected output for the optimization strategy could be the most suitable configuration of VMs to run a specific workload. In this work, is developed an Spark application based on the MapReduce model, named *Diff Sequences Spark*, which performs comparisons of biological sequences and identifies the occurrences of mismatching nucleotide characters. Such an application runs considering the SARS-CoV-2 coronavirus sequence comparisons, which is the virus responsible for the COVID-19 disease, using Amazon's AWS EC2 cloud service in both on-demand (standard) and spot (preemptive) VM instances. From the perspective of execution time and monetary cost optimizations, the adaptation of an execution cost model extracted from the literature is provided, whose experimental evaluation obtained low error rates. Experimental results using such optimizations outperformed scenarios where an inexperienced cloud user would select VMs without any reasonable criteria. Finally, reduced monetary costs were achieved when using spot instances compared to their respective on-demand options, even in scenarios with multiple spot Workers revocations on a Spark cluster.

Keywords: Apache Spark; MapReduce; Cloud computing; Optimization.

List of Figures

List of Tables

List of Abbreviations and Acronyms

AIMMS Advanced Interactive Multidimensional Modeling System

AMPL A Mathematical Programming Language

API Application Programming Interface

ARM Advanced RISC Machine

AST Abstract Syntax Tree

AWS Amazon Web Services

BFD Best Fit Decreasing

CaaS Compute as a Service

CLI Command-Line Interface

CoV Coronaviruses

CPU Central Processing Unit

CRESP Cloud RESource Provisioning

CSV Comma-Separated Values

DAG Directed Acyclic Graph

DNA Deoxyribonucleic Acid

DSA Distributed Sequence Alignment

DSL Domain-Specific Language

DW Distributed Write

EBS Elastic Block Store

EC2 Elastic Compute Cloud

EMR Elastic MapReduce

FIFO First In, First Out

GAMS General Algebraic Modeling System

GiB Gibibyte

GISAID Global Initiative on Sharing All Influenza Data

GPU Graphics Processing Unit

HaaS Hardware as a Service

HANA High-performance ANalytic Appliance

HDFS Hadoop Distributed File System

HP Hewlett-Packard

HPE Hewlett-Packard Enterprise

I/O Input/Output

IaaS Infrastructure as a Service

IBM International Business Machines

ILP Integer Linear Programming

IT Information Technology

JSON JavaScript Object Notation

JVM Java Virtual Machine

KB Kilobyte

KKT Karush-Kuhn-Tucker

LAN Local Area Network

LP Linear Programming

LSTM Long Short-Term Memory

MAE Mean Absolute Error

MapR-FS MapR File System

MERS Middle East Respiratory Syndrome

MILP Mixed-Integer Linear Programming

MIQCP Mixed-Integer Quadratically Constrained Programming

MIQP Mixed-Integer Quadratic Programming

MPL Mathematical Programming Language

MPLS Multi-Protocol Label Switching

MRA Multiple Regression Analysis

MRD Most Reference Distance

MSE Mean Square Error

MW Merged Write

NaaS Network as a Service

NCBI National Center for Biotechnology Information

NFSv3 Network File System Version 3

NNLS Non-Negative Least Squares

OS Operating System

PaaS Platform as a Service

QCP Quadratically Constrained Programming

QP Quadratic Programming

R-Squared Coefficient of Determination

RAM Random Access Memory

RDBMS Relational Database Management System

RDD Resilient Distributed Dataset

RMSE Root Mean Square Error

RNA Ribonucleic Acid

S3 Simple Storage Service

SaaS Software as a Service

SARS Severe Acute Respiratory Syndrome

SAS Statistical Analysis System

SD-WAN Software-Defined Wide Area Network

SIMD Single Instruction, Multiple Data

SLA Service-Level Agreement

SQL Structured Query Language

StaaS Storage as a Service

SW Smith-Waterman

TXT Text

USD United States Dollar

vCPU Virtual Central Processing Unit

VM Virtual Machine

VPN Virtual Private Network

XML Extensible Markup Language

YARN Yet Another Resource Negotiator

Contents

1 Introduction

1.1 Motivation

Over the last decades, the produced data volume reached an unprecedented level in academia and industry. According to Hey *et al.* (HEY et al., 2009), scientific advances will be increasingly driven by advanced computing capabilities, such as *Databases* and *Cloud Computing*. These resources will help the researchers manipulate and explore massive datasets, *i.e.*, obtain results through intensive computing for large volumes of data.

In 2020 each person on Earth produced about 1.7 MB of data every second (DOMO, 2020). Although this volume of data brings many analysis opportunities, *e.g.*, to discover hidden patterns, correlations, and preferences, it also leads to complex challenges, *e.g.*, how to process and query such volume of data from different sources and types and extract knowledge in a suitable time. The well-known and widely used data management approaches such as *Relational Database Management Systems* (*RDBMS*) usually do not scale for this heterogeneous volume of data (HU et al., 2014).

In recent years the database community has developed a series of solutions for large-scale data management. For instance, for the *NoSQL* approach, *MonetDB* (BONCZ et al., 2006), *MongoDB* (MAKRIS et al., 2021), and *Polystore systems* (KRANAS et al., 2021) are known solutions. However, the most successful approaches are the *Big Data* frameworks, such as *Apache Hadoop* (APACHE, 2022a) and *Apache Spark* (ZAHARIA et al., 2016). *Spark* improves the performance of applications by automatically exploiting parallelism and accomplishing in-memory data movement, in contrast to *Hadoop*, whose intermediate operations write to the storage volume. Besides, Spark allows users to develop their *Big Data* analytical applications without concerning the parallel processing environment complexity (PERERA et al., 2016).

Consider a real-world problem of bioinformatics: the biological sequences comparison. Nowadays, the *SARS-CoV-2 coronavirus* (CSC, 2020) sequence comparisons analysis is of great interest, as it allows understanding the behavior of the *COVID-19* disease. However, millions of *SARS-CoV-2 coronavirus* sequences are available in public genomic databases, *e.g.*, *NCBI* (NCBI, 2022a) and *GISAID* (GISAID, 2022). Commonly, the biological sequence comparisons occur using an alignment algorithm, but sometimes this approach is unwanted. For instance, Lau *et al.* (LAU et al., 2021) developed a rapid computational solution to identify the highly conserved regions of *SARS-CoV-2 coronavirus* sequences to index mutations across thousands of viral genomes without using any alignment method. Regardless of whether or not alignment methods are in use, few bioinformatics professionals have enough local computing power to perform a large number of biological sequence comparisons, which characterize a *Big Data* problem. Thus, the *Spark* framework is a reasonable choice to process that volume of data, as it can run on a single computer or a cluster of computers. In general, the *Cloud* environments are used to deploy a *Spark Cluster*. (YAN et al., 2016).

In *Cloud computing*, a range of physical and virtual resources can be dynamically allocated according to customer demand, often appearing limitless in terms of time or quantity. Of the deployment modes, *Public Clouds* are the most popular, with *Amazon Web Services* (33%), *Microsoft Azure* (21%), and *Google Cloud* (8%) the three most used service providers in the first quarter of 2022 (CHANNELE2E, 2022). Amazon Web Services (*AWS*) is considered the largest *Cloud computing* platform, spanning 84 availability zones in 26 geographic regions worldwide (AMAZON, 2022a) and offering 227 categories of services (AMAZON, 2022b), such as *Amazon Elastic Compute Cloud* (*EC2*) (AMAZON, 2022c). Besides the advantages of rapid provisioning of resources and significant reduction of the operational cost compared to dedicated infrastructures, *Cloud* providers also offer discounted prices for renting preemptible VMs. For example, the *Spot market* instances at *AWS EC2* can be up to 90% cheaper than their on-demand counterparts.

Although the above features represent a step forward, the current *Cloud* solution that promotes the usage of *Spark* is the *On-demand Managed Big Data Cluster* service. It is a *Platform as a Service* (*PaaS*) model that follows the pay-as-you-go pricing policy, *e.g.*, *Azure HDinsight*, *AWS EMR*, and *Google Dataproc*. Despite that, one key issue is how to select an optimized configuration of parameters, both at *Spark* and *Cloud* levels, to improve the application execution while not overspending financial resources. The *Cloud*-native *PaaS* solutions can scale the number of virtual machines (VAQUERO et al., 2013) but do not take into account the characteristics of a specific *Spark* application.

Different *Spark* applications may present diverse execution behaviors. So, the initial provisioning resources setting for a given *Spark* application on *Cloud* is still an open problem. A poor choice of parameters may significantly degrade the execution and lead to high financial costs that can make the application execution unfeasible. Many works propose solutions for optimizing the parameters at the *Spark* level (DE OLIVEIRA et al., 2021; ARMBRUST et al., 2015a; BOEHM et al., 2016). However, just a few (CHEN et al., 2014; ZHAO et al., 2015) focus on the initial provisioning of the environmental resources as a possible way of optimizing the execution of the *Spark* application.

1.2 Goals

One way to optimize the execution of a *Spark* application is to find the "best" estimation of resources, *e.g.*, the number of CPU cores of *VM*s in the *Spark Cluster*. Through the aid of a prediction model that estimates the runtime of a workload under certain conditions, supposed to be possible to estimate the necessary amount of resources to deploy in the *Cloud* and avoid unnecessary deployment before the beginning of the *Spark* application execution. It is worth mentioning that the resource selection may be tedious and error-prone if performed manually.

Having in mind the efficient run of *MapReduce*-like *Spark* applications on the *Cloud*, the main goal of this dissertation is to propose the formulation of the following optimization problems:

a. Runtime optimization subject to a budget constraint; and

b. Monetary cost optimization subject to a deadline constraint.

1.3 Contributions

The main contributions of this dissertation are:

I. Alternative proposals to run large-scale biological sequences comparisons using the *Diff Sequences Spark* application, a *MapReduce*-like *Spark* application that highlights the mismatching nucleotide characters occurrences without the use of conventional methods for alignment of biological sequences;

II. Practical directions to achieve a more stable and efficient *Spark* application execution and to avoid the *out-of-memory error* due to insufficient resources or the *task scheduling bottleneck* caused by a too short or too large amount of data processed;

III. A predictive model based on *Multiple Regression Analysis* (*MRA*) to assist the user in defining the ideal number of computational resources to be deployed in the *Cloud*;

IV. Mathematical formulations aiming to minimize the runtime and financial costs of *MapReduce*-like *Spark* applications in the *Cloud*;

V. Analysis of spot instances revocation scenarios, in terms of runtime and financial cost, when running the *Diff Sequences Spark* application in a *Spark Cluster* formed on *AWS EC2*; and

VI. Comprehensive evaluation of the optimization approaches.

1.4 Organization

The remainder of this dissertation is organized as follows. **Chapter 2** exposes the basic features of *Apache Spark*, the root aspects of *Cloud Computing*, and, in particular, of the *Amazon Public Cloud*'s *Infrastructure as a Service* solution (known as *AWS EC2*), the essential concepts and solving methods for *Optimization Problems*, and the principles and techniques of *Multivariate Analysis*. **Chapter 3** presents, in detail, the *MapReduce*-like *Spark* application, named *Diff Sequences Spark*, implemented for this study. **Chapter 4** introduces the related work. **Chapters 5** and **6** depict respectively the runtime prediction model and the mathematical model to optimize the runtime and financial costs of the *Diff Sequences Spark* application execution on *AWS EC2*. **Chapter 7** presents the execution results obtained for the *Diff Sequences Spark* application with the optimized parameters compared to their arbitrary selection and the execution results with various spot instances revocation scenarios on *AWS EC2*. Finally, **Chapter 8** expounds on the conclusions and future directions.

2 Background

2.1 Apache Spark

This section presents the basic concepts, main features, and deployment modes of Apache Spark, a unified open-source engine for distributed data processing.

2.1.1 Basic Concepts and Features

Apache Spark (APACHE, 2022b) is a framework designed for optimizing different types of workloads that vary from batch to iterative parallel operations over large datasets. Spark executes the applications by chaining a series of operations, as described by Zaharia *et al.* (ZAHARIA et al., 2016), and tries to avoid the significant I/O overheads found in alternative big data frameworks, *e.g.*, *Apache Hadoop*. The main advantage of Spark is that it improves the performance of applications by accomplishing, whenever possible, in-memory data movements, which avoids reading data from and writing results back to files.

A Spark application is composed of a *Driver* and *Workers*. The Driver is a process that controls the execution. Workers are nodes where the data are processed, and each Worker may have several associated *Executors* processes, which in turn execute *Tasks* associated with a specific *Job*. The number of Executors in a Worker is commonly defined by the user, and some approaches do that automatically, as suggested by De Oliveira *et al.* (DE OLIVEIRA et al., 2021) and by Kulkarni & Ramanathan (KULKARNI; RAMANATHAN, 2022). However, it is not a simple task since the choice of parameters depends on the computing environment chosen to execute the Spark application.

One key advantage of Spark in comparison to other big data frameworks is its in-memory structures, such as *Resilient Distributed Datasets* (*RDDs*), presented by Zaharia *et al.* (ZAHARIA et al., 2012b), and *DataFrames*, presented by Armbrust *et al.* (ARMBRUST et al., 2015b). Both are in-memory collections of partitioned data instances

that can be processed in parallel. While RDDs are sets of objects representing data, DataFrames are collections of distributed data with the same named columns scheme, *i.e.*, DataFrames act as tables in relational databases like *PostgreSQL* and *Oracle*.

Besides the advantages of in-memory data processing, Spark also provides fault tolerance mechanisms. Two types of failures may occur in the context of a Spark application: i.) Driver failure and ii.) Worker failure.

i.) In the case of a Driver failure, the *SparkContext* object (*i.e.*, the entry point to a Spark application) becomes unavailable, and all Executors lose their in-memory data. By defining a replicated Driver it can be solved, but it also adds some overhead to the application;

ii.) In the case of a Worker failure, all Executors associated with that Worker are lost, together will all their in-memory data. However, the data are commonly replicated to other Worker nodes, and each RDD has the capability of handling Worker failures. It is possible since Spark creates a logical execution plan (*i.e.*, *lineage graph*) for all the Tasks executed in the context of an application. For example, if a Worker fails during the application execution and an RDD is lost, then Spark can apply the computation originally done on that Worker following the lineage graph.

Spark also offers a set of libraries that can be seamlessly combined to achieve large-scale data processing complex capabilities, such as:

- *Spark SQL* for structured data processing, as proposed by Armbrust *et al.* (ARMBRUST et al., 2015b);
- *Spark Streaming* for real-time data stream processing, as proposed by Zaharia *et al.* (ZAHARIA et al., 2012a);
- *Spark MLlib* for distributed model training using machine learning techniques, as proposed by Meng *et al.* (MENG et al., 2016); and
- *Spark GraphX* for graphs and graph-parallel computation, as proposed by Xin *et al.* (XIN et al., 2013).

Spark provides an interactive shell tool (named *spark-shell*) for interactive data analysis and an *API* for self-contained applications, both available in various programming languages (*Scala*, *Java*, *Python*, and *R*). The Spark applications can be executed in a single computer, in an interconnected set of computers (*i.e.*, in-house cluster), or a shared on-demand computing resources delivered through the internet (*i.e.*, cloud computing).

2.1.2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs), the primary user-facing low-level API in Spark, are immutable collections of data objects logically distributed among the Workers, allowing them to be processed in parallel. RDDs can be created by:

i.) Parallelizing an existing data collection inside the Spark application; or

ii.) Referencing a dataset in an external storage system (*e.g.*, *Hadoop HDFS* and *Amazon S3*).

Due to their immutability, the result of processing an RDD can be a child RDD (intermediate result) or data object output (*e.g.*, write the final results to disk). In the first case, the child RDD always points to its parent RDD, keeping a kinship reference, *i.e.*, lineage concept. Thus, Spark can track the entire RDDs lineage and provide a fault-tolerant mechanism to recompute lost data in case of failures. RDDs can be cached on memory (default), disk, or both, enabling data reuse and execution of partial checkpoints during the data processing to avoid redundant computation.

In particular, RDDs are split into subsets, named *Partitions*, and distributed over the Worker nodes. An RDD holds a reference to partition objects and each partition object references a subset of the data. Partitions are assigned to Executors, and each is, by default, loaded in RAM. The number of partitions of an RDD depends on the way it is generated. Using the *parallelize* function to an existing data collection, Spark considers the total number of Executors cores from all alive Workers in the cluster (also known as *default parallelism*) and sets one partition per core. Using the *read* function to an external dataset, Spark splits the input file into chunks and sets one partition per file chunk, considering the file system's default data block size where the input file is stored. For instance, 128 MB data block size for *HDFS*.

All the Spark operators are classified in *Transformations* and *Actions*. A *Transformation* produces a new RDD from an existing one, whereas an *Action* enables the modification of current datasets without generating new RDDs. Spark evaluates RDDs lazily, *i.e.*, it does not compute their result immediately. Instead, Spark registers that a transformation is chained to an RDD and process it only when an action is invoked.

- Transformations can be classified as *Narrow* or *Wide*.

  i.) Narrow transformations (*e.g.*, *map* and *filter*) do not require rearranging data between partitions because each partition of the parent RDD is used by at most one partition of the child RDD, so they are stacked together and performed in parallel over different partitions;

  ii.) Wide transformations (*e.g.*, *join* and *reduceByKey*), on the other hand, require rearranging data between partitions since multiple child RDD partitions may depend of each parent RDD partition (data dependency). Thus, Spark performs a shuffle by moving data across the cluster with a new set of partitions.

- Actions (*e.g.*, *reduce*, *collect*, and *saveAsTextFile*) are operations that trigger data processing, previously defined by transformations, to produce result values instead of RDDs.

2.1.3 DataFrames

DataFrames, the focal abstraction in Spark's SQL API, are distributed collections of rows with the same schema (*i.e.*, typed columns that describe the structure of the data). DataFrame is similar to a table in a relational database and to an RDD of Row objects and can be manipulated similarly to RDDs. DataFrames can be created by:

i.) Converting an existing RDD within the Spark application;

ii.) Parallelizing an existing data collection within the Spark application; or

iii.) Referencing a dataset from an external storage system (*e.g.*, *Hadoop HDFS* and *Amazon S3*).

A Dataframe derived from a converted RDD will have the same amount of partitions as the source RDD. For the remaining, the logic described for the RDDs creation is still

applicable. Concerning the range of dataset sources available for the DataFrames creation, it is possible to load structured data files (*e.g.*, *Avro*, *Parquet*, and *Kafka*), semi-structured data files (*e.g.*, *CSV*, *JSON*, and *XML*), unstructured data files (*e.g.*, *TXT*), and tables (*e.g.*, *MySQL*, *DB2*, *Hive*, and *HBase*).

DataFrames keep track of their schema and automatically store data in a columnar format. It supports all relational operations like projection, join, filter, and aggregations (*e.g.*, *select*, *join*, *where*, and *groupBy*), using a domain-specific language (DSL) similar to *R*'s built-in data frames and *pandas*, which leads to more optimized execution. DataFrames can be more convenient and efficient than Spark's procedural API in many situations, for example, when it is needed to compute multiple aggregates in one pass using a SQL statement, a task hard to express in functional APIs.

DataFrames are also lazily evaluated, so each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until an output operation is invoked (*e.g.*, *show* and *write.csv*). Apart from the relational DSL, DataFrames can be registered as temporary tables in the system catalog and queried using SQL so optimizations can happen across SQL and the original DataFrame expressions.

The DataFrame operations build up an Abstract Syntax Tree (AST) of the relational expression, which then goes through an extensible query optimizer named *Catalyst*. It offers a general framework focused on optimizing the query processing with several sets of rules that handle different phases of the query execution: analysis, logical optimization, physical planning, and code generation.

2.1.4 Directed Acyclic Graphs

Directed Acyclic Graphs (DAGs) are logical representations of the workflow associated with the steps of a given Spark application, where *Vertices* represents RDDs and *Edges* the operations to be applied on RDDs.

When a Spark application is launched, the *DAGScheduler* translates the RDDs operations identified in the source code into a DAG whenever an action is called. Thus, a DAG consists of all transformations performed over RDDs to reach a particular result. Each action defined in the application spawns a *Job* composed of a *Stages* set. The DAGScheduler implements stage-oriented scheduling, where each stage groups neighboring transformations that can be executed together in parallel, *i.e.*, without rearranging data between the partitions of an RDD. A new stage is created whenever a shuffle operation (*i.e.*, a

wide transformation) has to be executed. The Stages that are not interdependent may be submitted to the cluster for concurrent execution if there are idle computing resources. Thus, the DAGScheduler determines a minimal number of stages to execute the job. This minimal Stages set is submitted as one *TaskSet* to the *TaskScheduler*, which is responsible for assigning the *Tasks* (units of processing) to the *Executors* that belong to the *Workers* nodes through the *Spark Cluster Manager*.

Spark adopts the data locality principle to execute tasks where the data is available, reducing the costs associated with data transferring:

- Firstly it searches for $PROCESS\_LOCAL$ tasks, which are the ones launched in the same Executor process (*i.e.*, data and tasks located together);
- If not, it checks for $NODE\_LOCAL$ tasks that may be in different Executor processes of the same Worker node;
- If not, it searches for $RACK\_LOCAL$ tasks, of which data and tasks are located in different worker nodes, requiring data transferring through the network;
- Finally, it searches for ANY tasks, which are any pending tasks that may execute in the target Worker.

2.1.5 Deployment Modes

Although Spark can run locally on one machine, it is essentially a distributed computational framework that allows building a scalable system for analytics over a large amount of data using a computer cluster. Each Spark application is a user program built on the *Spark API*, which consists of a set of *Transformations* and *Actions*. Once the application is submitted to the cluster, the *Driver program* is spawned. The Driver is a Java Virtual Machine (JVM) coordinator process responsible for executing the main function and creating the *SparkContext* object of a particular application. It can be launched in one of the following modes: *Client* or *Cluster*.

In the **Client Deploy Mode**, the Driver is deployed locally as an external client to the cluster (local machine). This mode allows the user to monitor the status of a particular application as its output is attached to the client's command-line interface (CLI), which is suitable for debugging or testing purposes. The application fails in case of a communication problem with the client since the Driver resides outside the cluster.

In the **Cluster Deploy Mode**, the Driver is deployed inside the cluster. This mode is suitable for production-ready applications with reliable performance and behavior. It minimizes the communication latency between the Driver and Executors since the application is not submitted from a machine physically far from the cluster. It is also advised for a long-term execution when monitoring is not needed (*Fire-and-forget* approach).

Figure 1 illustrates Spark's basic distributed architecture, which is described below.

- The *SparkContext* connects to the *Cluster Manager*, which is responsible for the resources allocation, and grants exclusive *Executors* for a particular application;
- *Executors* are JVM processes running in *Workers* (cluster nodes) that execute *Tasks* and may store data in memory (local cache) or disk. The number of Executors can be statically allocated (at the beginning of the application) or dynamically allocated (through the *Dynamic Resource Allocation* mechanism, based on the workload);
- *Tasks* are processing units (*i.e.*, operations over a specific *Partition* of the RDD) that are serialized and sent by the SparkContext to the Executors. Executors are in charge of the deserialization and the execution of each task. The serialization process transforms an object (combination of variables, functions, and data structures with state and behavior) into a format containing a series of bytes that can be transmitted faster over the network and reconstructed later. The Tasks are assigned to the idle Executors, which execute them in parallel according to the number of cores.

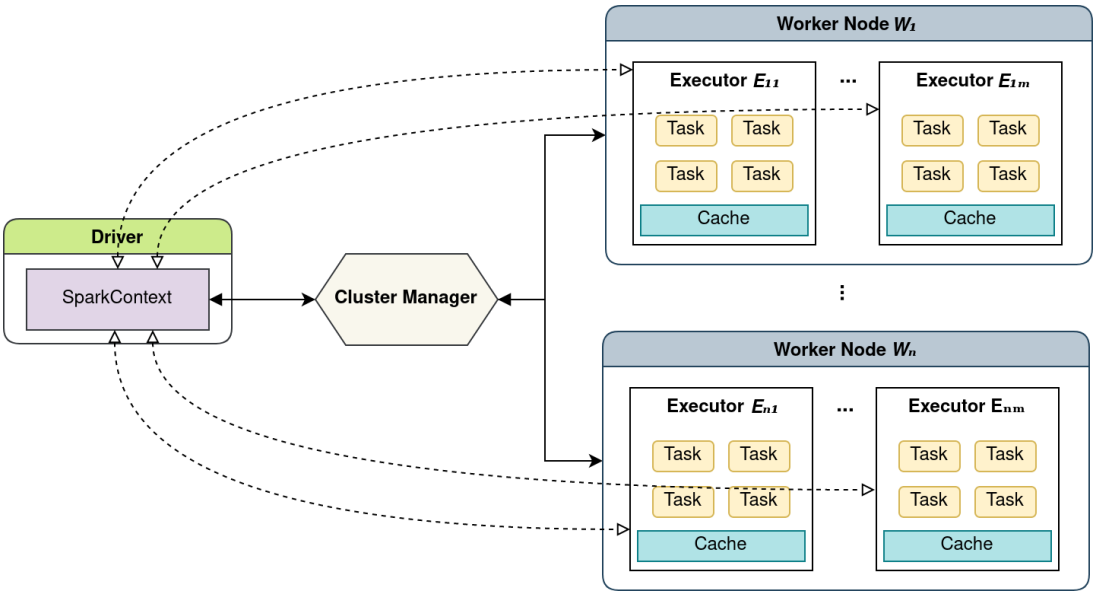


Figure 1: *Spark*'s basic distributed architecture.

This architecture isolates concurrent application executions in the cluster, as each Driver schedules its set of Tasks to its Executor processes set. The SparkContext is available throughout the lifetime of the application.

The classes of Cluster Manager supported by Spark for distributed environments are:

i.) **Standalone**, a simple cluster manager already included in Spark;

ii.) **Apache Mesos** (HINDMAN et al., 2011), a general cluster manager that can also run Hadoop MapReduce and service applications;

iii.) **Hadoop YARN** (KARANASOS et al., 2018), the resource manager, introduced in Hadoop 2.0, which splits up the functionalities of resource management and job scheduling/monitoring into separate daemons;

iv.) **Kubernetes** (BERNSTEIN, 2014), an open-source system to orchestrate containerized applications.

2.2 Cloud Computing

This section presents the root aspects of Cloud Computing, aiming at the service and deployment models with their advantages and concerns.

2.2.1 Definition and Characteristics

According to Foster & Gannon (FOSTER; GANNON, 2017), *Cloud Computing* is an on-demand computing and storage access model that delivers, through the internet, a larger capacity than is available locally. Accessing this capacity in the cloud may be cheaper, faster, and more convenient than acquiring and operating own computing and storage systems.

As a self-service model, the customer allocates computing resources automatically without requiring human interaction with the provider. The resources can be accessed via heterogeneous client platforms (such as workstations, laptops, and mobile phones) and are combined to serve multiple consumers at once due to the multi-tenant scheme used by the providers. Physical and virtual resources range can be dynamically assigned to customer needs. Generally, there is no control or knowledge over the actual location of the provided resources despite the possibility to specify higher-level abstracted locations

for them, *e.g.*, data center and region. Resources are promised to be provisioned to scale or release rapidly and often appear unlimited in terms of time or quantity. Their usage is monitored and controlled, providing cost transparency for providers and customers of the requested service.

2.2.2 Deployment Modes

Mell & Grance (MELL; GRANCE, 2011) presented four deployments modes for the cloud computing model:

I. **Private Cloud**: provisioned for exclusive use by a single organization comprising multiple business units consumers. *HPE Helion Managed Private Cloud*, *VMware vRealize Suite Cloud Management Platform*, *Dell Enterprise Private Cloud Solution*, *Cisco ONE Enterprise Cloud Suite*, and *AWS Virtual Private Cloud* are some options for this deployment mode;

II. **Community Cloud**: provisioned for exclusive use by a specific community of consumers from organizations with shared concerns, *e.g.*, mission, security requirements, and policy. *Cloud4C*, *Salesforce Community Cloud*, *IBM Cloud for Government*, and *Microsoft Government Community Cloud* are some options for this deployment mode;

III. **Public Cloud**: provisioned for open use by the general public. *Amazon Web Services*, *Microsoft Azure*, *Google Cloud*, *IBM Cloud*, and *CloudFlare* are some options for this deployment mode; or

IV. **Hybrid Cloud**: composed of two or more distinct cloud infrastructures that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability, *e.g.*, cloud bursting for load balancing between private and public clouds. *HPE Helion*, *IBM Bluemix*, *Verizon Enterprise*, *Fujitsu Hybrid Cloud Services*, and *Cisco Intercloud Fabric* are some options for this deployment mode.

2.2.3 Service Layers

The cloud environment can be a combination of services stacked into layers, having one service belonging to a higher layer composed of one or more services of its underlying layer.

The cloud computing systems fall into one of the following general layers: *Application*, *Software Environment*, *Software Infrastructure*, *Software Kernel*, or *Hardware/Firmware*. **Figure 2** illustrates the classic stack of cloud services (YOUSEFF et al., 2008).

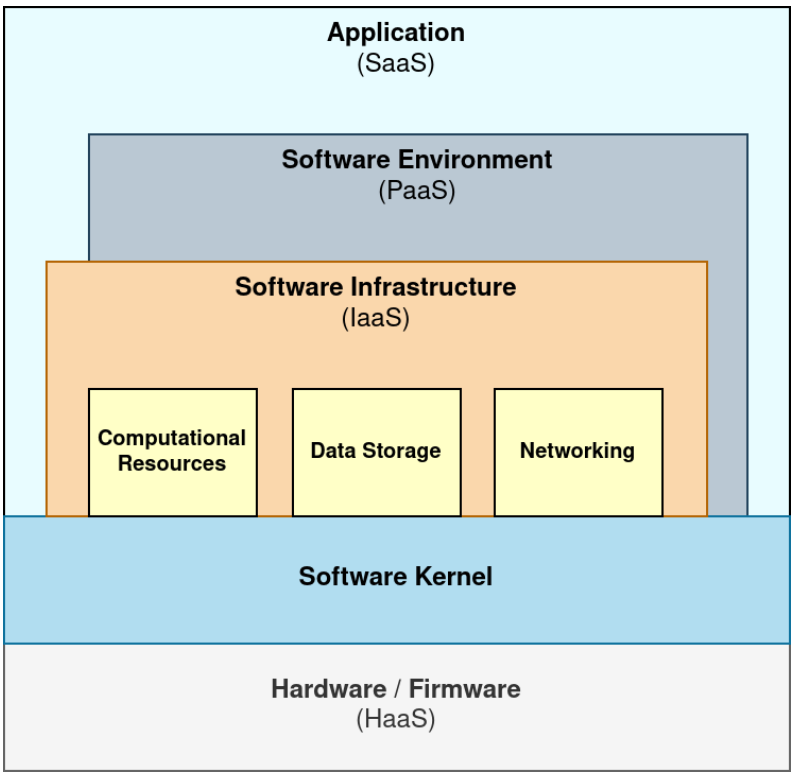


Figure 2: *Cloud computing*'s classic services stack.

The **Application Layer**, referred to *Software as a Service (SaaS)*, is the most commonly utilized layer of the cloud and offers many advantages, such as reduced costs for ongoing operations and software maintenance. The computational work is moved from the user environment into the provider's data centers, where the applications are deployed, allowing a superior performance without needing expensive equipment acquirement. Some limitations or concerns of using this layer are:

- Low customization capabilities for applications: the customers may be limited to specific functionality, performance, and integration as offered by the providers;
- Performance and downtime: maintenance, cyber-attacks, network issues, and system failures may impact the application availability;
- Data security and confidentiality: the transferring of sensitive business information may result in compromised security and compliance in addition to a cost for migrating large data workloads; and

- Interoperability: if the application is not designed for open integration standards, customers are forced to build their integration systems or reduce application dependencies.

Google Workspace, *Dropbox*, *Salesforce*, *Cisco WebEx*, and *GoToMeeting* are examples of SaaS solutions.

The **Software Environment Layer**, referred to *Platform as a Service (PaaS)*, is the most used by cloud applications developers. The providers supply a programming-language-level environment with a set of APIs to ease the interaction with other cloud applications, accelerate the development/deployment task, and support scalability and load balancing of applications.

This model is delivered via the web, so the developers do not have to worry about managing the infrastructure, software, or operating systems. It is even possible to build PaaS-integrated applications that use some of its software components, the so-called middleware applications. Some limitations or concerns of using this layer are:

- Operational limitation: may limit operational capabilities for end-users, affecting how the solutions are managed, provisioned, and operated (*e.g.*, customized operations with management automation workflows);
- Compatibility with legacy systems: PaaS may not be a plug-and-play solution with existing legacy applications and services, so several customizations and configuration changes may be necessary for legacy systems to work with this service, resulting in a complex and expensive IT system; and
- Runtime limitations: systems built using this model may not be optimized for the desired programming language, or the customers may not be able to develop custom dependencies for their solutions.

Google App Engine, *AWS Elastic Beanstalk*, *Heroku*, *Red Hat OpenShift*, and *Azure Windows Server* are examples of PaaS solutions.

The **Software Infrastructure Layer**, referred to *Infrastructure as a Service (IaaS)*, provides automated compute resources to other higher-level layers (*e.g.*, SaaS and PaaS) for constructing new cloud applications or software environments. The focal services offered in this layer are computational resources, data storage, and network delivered

through virtualization technology. They are typically set up by the customer through a web dashboard or an API, giving complete control over them.

This model provides similar capabilities and technologies as an in-house data center without requiring the user to maintain or manage it physically. As opposed to SaaS or PaaS, the IaaS clients are responsible for managing aspects, such as operating systems, applications, and data. The cost of this service is associated with the resources' total use time, which can be purchased as needed and are dynamic, flexible, and highly scalable. Some limitations or concerns of using this layer are:

- Multi-tenant isolation security: the provider has to ensure that previous customers' data cannot be accessed by newer since the resources are dynamically allocated across users as they are made available;
- Learning stage: the customers are responsible for resources monitoring and management, backup, and data security, so training may be required for the staff to learn how to safely and effectively use it; and
- Host infrastructure security: system vulnerabilities can still be sourced from the host (*e.g.*, execution of arbitrary code on the host with the privileges of the hypervisor process) or other VMs (*e.g.*, guest-to-guest attacks, in which attackers use one VM to access or control other VMs on the same hypervisor) and may expose data communication in this model to unauthorized entities.

A. The *Computational Resources* subcategory, referred to *Compute as a Service (CaaS)*, provides computing units, of which VMs are the most common. It offers finer-granularity flexibility as the user gets super-user access to them, allowing customized software stack for performance and efficiency needs. This cloud component grants the user an unprecedented range of settings and protects the provider's physical infrastructure. However, there is a performance interference between VMs sharing the same physical node, which often prevents providers from guaranteeing full performance to their clients. Usually, a performance decrease is noted if compared with a dedicated environment. It may also impact the performance of the upper layers, affecting the SLAs built on top of IaaS's SLA. *DigitalOcean Droplets*, *Google GCE*, *Amazon EC2*, *IBM Cloud Virtual Servers*, *Linode*, and *Azure Virtual Machines* are examples of CaaS solutions.

B. The *Data Storage* subcategory, referred to *Storage as a Service (STaaS)*, provides a remote repository and access to the user's data. It offers cloud applications data

scalability beyond the servers' limited storage resources. Stringent requirements are expected for the data storage systems, such as high availability, reliability, performance, replication, and data consistency. However, no environment features all of them due to conflicting goals. The STaaS providers usually implement their system to favor one feature over the others, indicating their choice through respective SLA. *Azure Disk Storage*, *Google Cloud Storage*, *Amazon S3*, *Oracle File Storage*, and *IBM Cloud Object Storage* are examples of STaaS solutions.

C. The *Networking* subcategory, referred to *Network as a Service (NaaS)*, allows the customers to build and operate network architectures without maintaining their infrastructure. This model can replace virtual private networks (VPNs), multiprotocol label switching (MPLS) connections, legacy network configurations, and LAN hardware (*e.g.*, firewall devices and load balancers for distributing network traffic across multiple servers). It implements communications features for the cloud systems, such as network security, dynamic provisioning of virtual overlays for traffic isolation or dedicated bandwidth, communication encryption, and network monitoring. It assures communication that is service-oriented, configurable, schedulable, and reliable. *Aryaka SD-WAN*, *Pertino Cloud Network Engine*, and *Akamai Content Delivery Network* are examples of NaaS solutions.

The **Software Kernel Layer** provides the core software management for the physical servers that compose the cloud environment. It can be implemented as an OS kernel, hypervisor, cluster middleware, or virtual machine monitor.

The **Hardware/Firmware Layer**, referred to *Hardware as a Service (HaaS)*, provides the physical components and switches that form the backbone of the cloud. Large companies with massive computational requirements are the target customers of this service and take advantage of this model by not investing to acquire and maintain their data centers. HaaS carries the benefit of built-in maintenance and troubleshooting costs. Service provisioners have the technical expertise to host, operate and cost-effectively upgrade the infrastructure. Upgrades consider the lifetime of the sublease by the customer.

Many technical challenges are addressed to the providers, such as efficiency, ease, and speed of large-scale systems provision, data center management, scheduling, and power-consumption optimizations. Since customers do not own the IT equipment outright, it is necessary to ensure reliable contracts with the service providers as any non-payment can lead to equipment shutdown. *AWS Outposts*, *Azure Stack*, and *Google Anthos* are examples of HaaS solutions.

2.2.4 Amazon EC2

Elastic Compute Cloud (EC2) is an IaaS public cloud solution hosted by Amazon Web Services (AWS) that offers a broad compute platform with over 500 virtual machine instance types, according to Amazon (AMAZON, 2022c). The instances are optimized and suited to different use cases. The following types are available for the customers to execute applications and workloads with optimal compute, memory, storage, and networking balance:

I. **General Purpose**, which is often used for (AMAZON, 2022d):

   * Development of macOS applications (*Mac* family);
   * Websites, web applications, microservices, code repositories, and other lighter processing tasks (*fT2*, *T3*, *T3a*, and *T4g* families);
   * Small and mid-size databases, enterprise applications, low latency networking, simulation modeling, telecommunication industries, and gaming servers (*M4*, *M5*, *M5a*, *M5n*, and *M5zn* families); and
   * Backend servers supporting enterprise applications, application development environments, and mid-size data stores (*A1*, *M6a*, *M6i*, and *M6g* families).

II. **Compute Optimized**, which is often used for (AMAZON, 2022d):

   * High-performance web servers, batch processing, distributed analytics, highly scalable multiplayer gaming, video encoding, scientific modeling, high-performance computing, log analysis, and CPU-based machine learning inference (*C4*, *C5*, *C5a*, *C5n*, *C6a*, *C6g*, *C6i*, *C6gn*, and *C7g* families); and
   * High-performance computing workloads like fluid dynamics, molecular dynamics, and weather forecasting or high-performance network workloads (*Hpc6a* family).

III. **Memory Optimized**, which is often used for (AMAZON, 2022d):

   * Memory-intensive workloads, high-performance databases such as SAP HANA, SQL, and NoSQL, distributed web scale in-memory caches such as KeyDB, Memcached, and Redis, mid-size in-memory databases, real-time big data analytics such as Apache Hadoop and Apache Spark, data mining, and real-time processing of unstructured data (*R4*, *R5*, *R5a*, *R5b*, *R5n*, *R6g*, *R6i*, and *Z1d* families); and

* Electronic design automation workloads such as physical verification, static timing analysis, power signoff and chip gate-level simulation, in-memory analytics such as SAS and Aerospike, and big data processing engines such as Presto (*X1*, *X1e*, *X2gd*, *X2idn*, *X2iedn*, and *X2iezn* families).

IV. **Storage Optimized**, which is often used for (AMAZON, 2022d):

* I/O intensive workloads that require higher storage density and fast access to data sets on local storage, NoSQL databases such as Apache Cassandra and MongoDB, data warehousing, data streaming, and analytics workloads (*I3*, *I3en*, *I4i*, *Im4gn*, and *Is4gen* families); and
* Distributed file systems such as HDFS and MapR-FS, Big Data analytical workloads such as Apache Hadoop and Apache Spark, massively parallel processing data warehouses such as Redshift and HP Vertica, data processing applications such as Apache Kafka, multi-node file storage systems such as Lustre and BeeGFS, MapReduce-based workloads and network file systems such as NFSv3 (*H1*, *D2*, *D3*, and *D3en* families).

V. **Accelerated Computing**, which is often used for (AMAZON, 2022d):

* Machine learning, high-performance computing, computational fluid dynamics, computational finance, seismic analysis, molecular modeling, genomics, rendering, speech recognition, and other server-side GPU compute workloads (*P2*, *P3*, and *P4* families);
* Deep learning training, object detection, image recognition, natural language processing, forecasting, image, and video analysis, search, ranking and recommendation engines, advanced text analytics, and language translation and transcription (*DL1*, *Inf1*, and *Trn1* families);
* 3D visualizations, application streaming, graphics-intensive applications such as remote workstations, video rendering, and transcoding, autonomous vehicle simulations, and cloud high fidelity graphics real-time gaming (*G3*, *G4ad*, *G4dn*, *G5*, and *G5g* families); and
* Genomics research, financial analytics, real-time video processing, Big Data search and analysis, and live event broadcast (*F1* and *VT1* families).

Many of these types also offer *bare metal* (dedicated) instances, *e.g.*, *m6i.metal*, the corresponding option of the *m6i.32xlarge* instance type. The bare metal instances allow

direct access to the processor and memory of the underlying physical server for running applications in non-virtualized environments or with a hypervisor implemented by the customer. This option provides high-performance computing with highly customizable hardware resources that can be tuned up for maximum performance and provide a dedicated compute environment with more security (*e.g.*, data privacy by being the only tenant on the server).

Regarding the instances billing in EC2, which can be per second, hour, or month of use depending on the instance type, pricing market, and location region, the main options offered are *On-Demand*, *Reserved*, and *Spot*:

I. **On-demand instances** are recommended for applications with short-term or unpredictable workloads that cannot be interrupted. Also, it is suitable for applications under development or test due to its high availability and reliability and for customers that prefer its flexibility without any up-front payment or long-term commitment. The user pays for the period the instance is under rental. However, this option has a higher cost than the other pricing models;

II. **Reserved instances** are recommended for customers that can commit for a 1 to 3-year term, have consumption predictability, or their applications require reserved capacity for a long and stable execution. The reserved instances option can provide a significantly higher discount than the on-demand market, depending on the term length and the upfront payment; and

III. **Spot instances** are recommended for fault-tolerant workloads that are flexible to run on multiple instance types such as batch jobs, containerized workloads, web applications, Big Data clusters, and High-Performance Computing clusters. Compared to the other pricing models, the spot provides the highest discount (up to 90% cheaper than on-demand) but with a fluctuating availability and occasionally pricing volatility since the customers are bidding on spare computing capacity, *i.e.*, idle EC2 instances available. The users must define the maximum price they are willing to pay per hour per instance, which can be lower, equal, or higher than the current price for a particular type. If the customer bid is lower than the current price (gradually adjusted by Amazon), then the instance is not launched.

Amazon EC2 can interrupt the acquired spot instance when: the demand for spot instances rises, the supply of spot instances decreases, or the spot price exceeds the customer bid (maximum price), so the SLA's aspects may be seriously affected by using

this pricing model. The customer may set one of the following interruption behavior to the spot instances: *Hibernate*, *Stop*, or *Terminate* (default).

- If **Hibernate** is defined, only Amazon EC2 can hibernate and resume a hibernated instance. The data from memory (RAM) is saved to the EBS root volume, the EBS root device and attached EBS volumes are preserved, and their data is persisted. The wake-up occurs when the capacity is available for the same instance type as the hibernated instance in the same availability zone. Upon restart, the EBS root device is restored from its earlier state, including the RAM content, previously data volumes are reattached, and the instance retains its instance ID;
- If **Stop** is defined, only Amazon EC2 can restart a stopped instance. The EBS root device and attached EBS volumes are preserved, and their data is persisted. The restart process occurs when the capacity is available for the same instance type as the stopped instance in the same availability zone. Upon restart, the EBS root device is restored from its earlier state, previously attached data volumes are reattached, and the instance retains its instance ID; and
- If **Terminate** is defined, Amazon EC2 terminates the instance immediately, wiping all data and settings (by default).

As pointed out by Teylo *et al.* (TEYLO et al., 2021a) and by Brum *et al.* (BRUM et al., 2021), there are many opportunities and benefits (*e.g.*, monetary cost reduction) in using spot instances. For example, when the application can handle failures or is executed within a framework that provides recovering mechanisms in case of VM revocations.

2.2.4.1 Typical Spark Cluster Deployment Alternatives on AWS

The AWS services typically used for Spark clusters deployment are Elastic Compute Cloud (EC2) and Elastic MapReduce (EMR).

Elastic MapReduce (EMR) is a PaaS solution that simplifies building and operating Big Data environments and applications through automatically cluster infrastructure provisioning, scaling, and reconfiguring, according to Amazon (AMAZON, 2022e). One of the ways to use EMR is to build a Spark cluster from EC2 instances, where the customers choose what instance types to the provision based on their application's requirements. So, EMR is suitable for beginner cloud users due to its automatic management and preconfigured environments. However, there is a cost associated with using the EMR service.

In particular, for each EC2 instance, users are additionally charged per second from the moment their cluster starts until it is terminated, with a one-minute minimum. **Figure 3** illustrates the main pricing components of EMR, considering the monthly cost in USD for using five EC2 instance types (*c5.xlarge*, *m5.xlarge*, *r5.xlarge*, *i3.xlarge*, and *p2.xlarge*) supported in the Northern Virginia region (*us-east-1*). The average fee for the mentioned instances in EMR is 25.06% beyond the usual EC2 charges. For all the EC2 instance types supported in Northern Virginia, the average additional cost is 18.72%, with a standard deviation of 8.26 (AMAZON, 2022f).

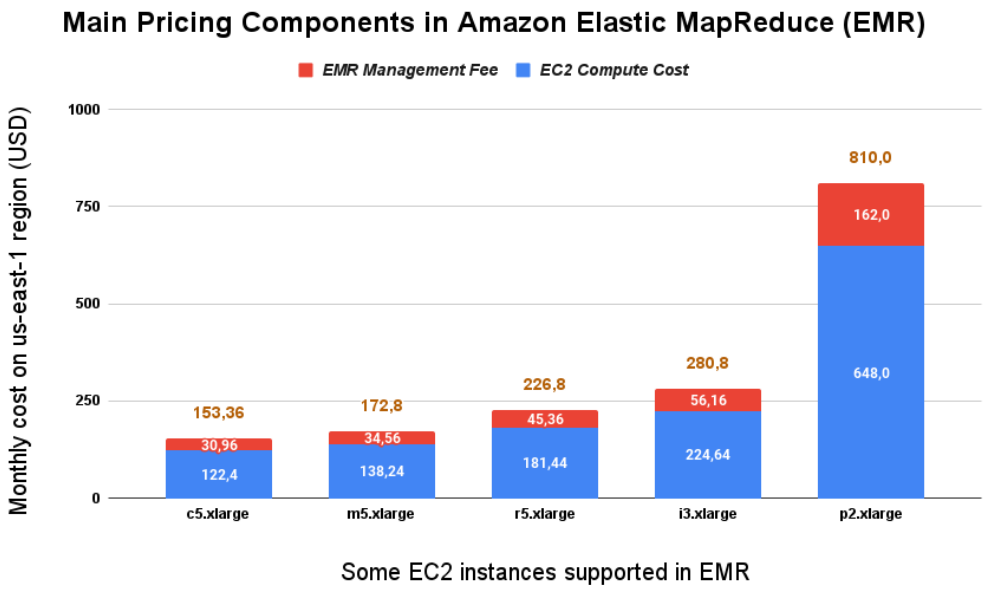


Figure 3: *EMR*'s main pricing components for some EC2 instances types.

Considering the additional cost of the EMR service, the Spark clusters deployed during this study were manually built using the EC2 instances. While this option requires cloud expertise, it supposedly offers cheaper and more customizable environments.

2.3 Optimization Problems

This section presents the basic definition, concepts, and solving methods for optimization problems, which find the best of all possible solutions that satisfy a given objective.

2.3.1 Definition

Optimization Problem, as presented by Papadimitriou & Steiglitz (PAPADIMITRIOU; STEIGLITZ, 1998), can be seen as the choice of the best configuration or set of parameters

to achieve a given goal. In the past few decades, a hierarchical collection of optimization problems has come out with corresponding solving techniques that are mainly iterative. Their convergence is studied through continuous mathematical analysis.

Equation 2.1 represents one end of this hierarchy: the *general nonlinear programming problem*. Let $\mathbb{R}^n$ be the n-dimensional real vector space (*i.e.*, a set of ordered n-tuples of real numbers), f be the solution or objective function, and g_i and h_j be the constraints functions (*i.e.*, conditions that f must satisfy). In this class of optimization problems, the goal is to find $x \in \mathbb{R}^n$, where f, g_i, and h_j are general functions of x. When f is convex, g_i is concave, and h_j is linear, the problem is defined as a *convex programming problem*. In this convex class, if f has a local minimum, it will also be a global minimum. When f, g_i, and h_j are all linear, it is defined as a *linear programming problem* (LP). In this linear class, any problem is reduced to a solution from among a finite set of possible solutions. Representative examples of linear problems are resource allocation problems, production problems, or network flow problems.

$$\begin{aligned} &\text{minimize} && f(x) && \\ &\text{subject to} && g_i(x) \geq 0, && i = 1, \ldots, m \\ & && h_j(x) = 0, && j = 1, \ldots, p \end{aligned} \tag{2.1}$$

There are two categories of optimization problems: the one with continuous variables and the one with discrete variables (combinatorial problem). In *continuous* ones, the goal is a set of real numbers or a function. In *combinatorial* problems, the goal is to find an object from a finite or possibly countably infinite set (typically an integer, set, permutation, or graph). Since these two categories often have different characteristics, their solving methods have become quite unlike. Linear programming, however, can fit in both, having a fundamental role in many strictly combinatorial problems.

Consider the mapping $c : F \to \mathbb{R}^1$, where F is any set (*i.e.*, the domain of feasible points) and c is the cost function. Stated the input data with enough information to obtain a solution, an instance of a general optimization problem is a pair (F, c), where we want to find $f \in F$ such that: $c(f) \leq c(y)$, $\forall y \in F$. Such f is called a globally optimal solution to the given instance (or simply an optimal solution). An optimization problem is an instance set I of an optimization problem usually generated similarly.

2.3.2 Forms of the Linear Programming Problem

Let A be a matrix $m \times n$ with rows a_i', $x \in \mathbb{R}^n$, M the set of row indices corresponding to equality constraints, $\bar{M}$ the set of row indices corresponding to inequality constraints, N the column indices corresponding to constrained variables, $\bar{N}$ the column indices corresponding to unconstrained variables, b an m-vector of integers, and c an n-vector of integers. Papadimitriou & Steiglitz (PAPADIMITRIOU; STEIGLITZ, 1998) defined the *General Form* of an LP as follows (**Equation 2.2**):

$$\begin{aligned} &\min c'(x) \\ &a_i'(x) = b_i, \quad && i \in M \\ &a_i'(x) \geq b_i, \quad && i \in \bar{M} \\ &x_j \geq 0, \quad && j \in N \\ &x_j \lesseqgtr 0, \quad && j \in \bar{N} \end{aligned} \tag{2.2}$$

Besides the general form, Papadimitriou & Steiglitz (PAPADIMITRIOU; STEIGLITZ, 1998) also presented the *Canonical Form* and the *Standard Form* of the linear programming problem, as expressed in **Equations 2.3** and **2.4**, respectively:

$$\begin{aligned} &\min c'(x) \\ &A(x) \geq b \\ &x \geq 0 \end{aligned} \tag{2.3}$$

$$\begin{aligned} &\min c'(x) \\ &A(x) = b \\ &x \geq 0 \end{aligned} \tag{2.4}$$

The canonical, standard, and general forms are all equivalent, so an instance in one form can be converted to another by a simple transformation such both have the same solution. This mathematical result is particularly crucial for the *Simplex Method* implementation proposed by Dantzig (DANTZIG, 1990), widely adopted to solve LPs, as it assumes that the LP is in standard form.

2.3.3 Integer Linear Programming Problem

Papadimitriou & Steiglitz (PAPADIMITRIOU; STEIGLITZ, 1998) defined *Integer Linear Programming* (ILP) as a particular case of LP in the standard form, as expressed in **Equation 2.5**, where A, b, c, and $x \in \mathbb{Z}$ (*i.e.*, assumes only integer values). ILP can be defined similarly in canonical or general form, as seen in **Subsection 2.3.2**, as they are equivalent forms.

$$\begin{aligned} &\min c'(x) \\ &A(x) = b \\ &x \geq 0, \qquad x \in \mathbb{Z} \end{aligned} \tag{2.5}$$

The need to formulate and solve ILPs came from the fact that in some LP applications, fractional solutions were undesirable (*e.g.*, an application in which x_j is the number of resources assigned to the receiver j). When it is expected for the solution x to contain large integers (that may be insensitive to rounding), one corresponding simple LP approach would be round the solution of ILP to the closest integer. However, rounding to a feasible integer solution may not always be straightforward. Most frequently, an ILP consciously uses integer values for modeling combinatorial constraints or different nonlinearities. These ILPs are not susceptible to the rounding approach because rounding defeats the purpose of the ILP formulation. Another unsuccessful attempt would be a local search in the neighborhood for the optimal continuous solution. The search space is often highly constrained, and searching around it does not produce the optimal solution for the ILP.

2.3.4 Optimization Methods

Rothlauf (ROTHLAUF, 2011) presented the goal of an optimization method: to find an optimal or near-optimal solution with low computational effort. The effort is measured as the time (*i.e.*, number of computation steps) and the space (*i.e.*, amount of computer memory) consumed by the method. There is a trade-off between solution quality and effort for many optimization methods, where an increasing effort typically provides a higher quality solution.

Arora & Barak (ARORA; BARAK, 2009) provided the broad notions of computability theory and complexity classes, defined in terms of the computational effort of solving the problems concerning particular computational resources like time or memory. Typically

the computer is assumed to be deterministic (*i.e.*, given the current state and any inputs, there is only one possible decision that can be taken by the computer) and sequential (*i.e.*, it performs the computation steps one after the other). The complexity class *P* represents the set of problems that can be solved in a deterministic sequential computer in time that is polynomial to the input size. The complexity class *NP* represents the set of problems whose solutions can be verified in polynomial time on a non-deterministic machine, given a certificate (also called a witness) that certifies the answer to a computation. The complexity class *NP-complete* represents the set of problems that any other NP problem can be reduced in polynomial time and whose solution may still be verified in polynomial time. So, if a deterministic polynomial-time algorithm can be found to solve one of them, then every NP problem is solvable in a polynomial time. The complexity class *NP-hard* represents the set of problems that any other NP problem can be reduced in polynomial time and whose solution does not have to be verifiable in polynomial time.

Rothlauf (ROTHLAUF, 2011) classified the optimization methods into two types: *Exact* and *Heuristic*. The **Exact methods** guarantee to find an optimal solution, opposite to the Heuristic methods. They are often chosen if it is possible to solve an optimization problem with the amount of effort that grows polynomially with the problem size (whose belong to the complexity class P). Some types of exact optimization methods are:

I. *Analytical and Numerical Optimization Methods*, used to find optimal solutions if the problem is well-defined and highly structured, the objective function is explicitly known, and there are no constraints on the decision variables; and

II. *Linear Optimization Methods*, used if the objective function depends linearly on the decision variables and all relations among the variables are linear. In *Continuous* optimization problems, objective function and constraints are continuous and linear. Two methods are broadly used for this subset of linear problems: *Simplex* and *Interior Point*. In *Discrete* optimization problems, decision variables are all integers. The methods commonly used for this subset of linear problems are *Decision Tree-based* search approaches (*e.g.*, breadth-first, depth-first, uniform-cost, best-first, and A*), *Branch-and-Bound*, *Dynamic Programming*, and *Cutting Plane*.

However, if the problem needs exponential effort, then even medium-sized problem instances become intractable and cannot be solved with exact methods. In this situation, the **Heuristic methods** are frequently used, showing good performance for many NP-complete problems and problems of practical relevance. Such methods are usually

problem-specific, as they exploit their particular characteristics. To efficiently build heuristics, some knowledge about the structure of the problem to be solved is required, as using a heuristic method incompatible with an optimization problem leads to low quality solution. Some types of heuristic optimization methods are:

I. *Heuristics*, which are often based on *Greedy Search* (*e.g.*, best-first), try to exploit problem-specific knowledge, but no guarantee of finding an optimal solution. In construction heuristics, also known as single-pass heuristics, a solution is built from scratch (*i.e.*, an empty solution) in a step-wise creation process. In each step, parts of the solution are fixed (*e.g.*, Nearest Neighbor, Nearest Insertion, Cheapest Insertion, and Furthest Insertion). In improvement heuristics, a complete solution (*i.e.*, initial solution) is used at the beginning, and improvement attempts are performed iteratively until the current solution can not improve anymore, *i.e.*, a local optimum has been reached (*e.g.*, Two-Opt, k-Opt, and Lin-Kernighan);

II. *Approximation Algorithms*, which are heuristics that allow developing bounds on the quality of the returned solution (*e.g.*, Fully Polynomial-Time Approximation, Polynomial-Time Approximation, and Constant-Factor Approximation); and

III. *Modern Heuristics or Metaheuristics*, which are extended variants of improvement heuristics, are general-purpose methods applicable to many problems, whose relevant elements are: the representation and variation operators used, the fitness function, the initial solution, and the search strategy. Metaheuristics use two phases during the search step: intensification and diversification. The intensification step improves the quality of existing solutions. The diversification step explores new areas of the search space by systematically modifying them and accepting partial or complete solutions that are inferior to the currently obtained solutions. Metaheuristics usually run a limited number of search steps and are stopped after a certain quality solution level is reached or several search steps are performed. Representative examples of metaheuristics are *Evolutionary Algorithms*, *Simulated Annealing*, *Tabu Search*, and *Ant Colony Optimization*.

2.3.5 Gurobi Optimizer

Gurobi Optimizer (GUROBI, 2022) is a state-of-the-art solver for mathematical programming models. It supports an extended list of lightweight APIs for modeling, including *Object-Oriented* (*e.g.*, C++, Java, .NET, and Python), *Matrix-Oriented* (*e.g.*, C, MATLAB, and R), and *Standard Modeling Languages* (*e.g.*, AIMMS, AMPL, GAMS, and MPL).

Gurobi grants a flexible deployment, allowing a user to solve a single model using one machine or many users to solve multiple models using many computers (Gurobi Compute Server). The models can also be solved locally or on a private or public cloud (Gurobi Instant Cloud). Besides the full-featured evaluation license for commercial users and free trial hours for the cloud environment, Gurobi supports research, optimization teaching, and use within academic institutions with free, full-featured copies.

The solvers included in Gurobi are designed to exploit modern architectures and multi-core processors, using advanced implementations of the latest algorithms to solve a range of optimization problems, such as *Linear Programming* (LP), *Mixed-Integer Linear Programming* (MILP), *Mixed-Integer Quadratic Programming* (MIQP), *Quadratic Programming* (QP), *Quadratically Constrained Programming* (QCP), and *Mixed-Integer Quadratically Constrained Programming* (MIQCP).

Regarding the implemented algorithms, *Simplex*, *Parallel Barrier with Crossover*, *Concurrent*, and *Sifting* are used for LP, and *Simplex* and *Parallel Barrier* are used for QP. These implementations for LP and QP quickly and robustly solve models with millions of variables and constraints. *Parallel Second-Order Cone Programming Barrier* is used for QCP, and *Deterministic Parallel*, *Non-Traditional Search*, *Heuristics*, *Solution Improvement*, *Cutting Planes*, and *Symmetry Breaking* are used for MIP (MILP, MIQP, and MIQCP). The MIP and barrier optimizers include shared-memory parallel algorithms that use all available cores and sockets, while the MILP and MIQP optimizers utilize an advanced branch-and-cut algorithm.

2.4 Multivariate Analysis

This section presents the basic definition, concepts, and techniques of multivariate analysis, whose objective is to provide statistical methods that convert collected information into decision-making knowledge.

2.4.1 Definition and Basic Concepts

According to Hair *et al.* (HAIR et al., 2018), *Multivariate Analysis* refers to all the statistical techniques that simultaneously analyze multiple measurements on individuals or objects under investigation, *i.e.*, different variables in a single relationship or set of relationships. Any simultaneous analysis of more than two variables can be loosely considered multivariate analysis. However, to be defined precisely as multivariate, all the variables must be random and interrelated such that their different effects cannot meaningfully be interpreted separately. Thus, the multivariate character lies in the multiple combinations of variables (known as variates) and not only in the number of variables or observations. The multivariate analysis introduces some concepts of particular relevance for the success of any technique: i) *Variate*, ii) *Measurement Scales*, and iii) *Measurement Error*.

Variate is the linear combination of variables specified by the researcher with weights empirically determined according to the multivariate technique selected to meet a specific objective. In the *Multiple Regression* technique, for example, the variate is determined to maximize the correlation between the multiple independent variables and the single dependent variable. It is represented as $V = w_1 \cdot X_1 + w_2 \cdot X_2 + w_3 \cdot X_3 + \ldots + w_n \cdot X_n$, where V corresponds to the variate value (result), n is the number of weighted variables, X_n is the observed variable, and w_n is the weight determined by the multivariate technique. The result is a single value, the combination of the whole set of variables that best achieves the objective.

Measurement Scales are used to accurately represent the concept of interest (*i.e.*, each variable used) and are essential for the suitable selection of the multivariate method. Based on the class of attributes, the data can be classified into nonmetric (qualitative) or metric (quantitative) categories.

- *Nonmetric Measurements* can be made with either a nominal or an ordinal scale, and they describe differences in type or kind by indicating the presence or absence of a characteristic or property (discrete features). The *nominal scales*, also known as categorical scales, provide the number of occurrences in each class or category of the variable studied (*e.g.*, demographic attributes such as religion or occupation). In the case of *ordinal scales*, variables can be ordered or ranked concerning the amount of the feature possessed, indicating the relative positions in an ordered series (*e.g.*, products ranking based on satisfaction level).

- *Metric Measurements* are appropriate for attributes involving amount or magnitude and are used when subjects differ in amount or degree on a particular characteristic. They can be made with either an interval or a ratio scale. Both *interval* and *ratio scales* have constant units of measurement, so differences between any two adjacent points on any part of these scales are equal. Nearly any mathematical operation can be performed on them. However, the interval scales use an arbitrary zero point (*e.g.*, Fahrenheit and Celsius temperature scales), whereas ratio scales include an absolute zero point (*e.g.*, weighing machines). Thus, it is unreasonable to say that any value on an interval scale is a multiple of some other point on the scale, *e.g.*, in the case of a ratio scale.

Measurement Error is the degree to which the observed values are not representative of the actual values, having sources of error ranging from the imprecision of the measurement to the inability of answerers to provide accurate information. So, all the variables used in multivariate techniques must be assumed to have some degree of measurement error, *i.e.*, having the valid value bound with measurement noise. The measurement error degree is assessed through two essential characteristics: *Validity*, the degree to which a measure accurately represents what it is supposed to, and *Reliability*, the degree to which the observed variable measures the genuine value and is error-free. The improvement of both characteristics results in a more accurate portrayal of the variables of interest, avoiding distortions in the observed relationships and making the multivariate techniques more effective.

2.4.2 Multivariate Techniques Types

Hair *et al.* (HAIR et al., 2018) emphasized that many multivariate analysis techniques are extensions of *Univariate* and *Bivariate*. The former is for single-variable distribution analysis and the latter to analyze two variables (*e.g.*, *Cross-Classification*, *Correlation*, *Analysis of Variance*, and *Simple Regression*). The simple regression with one predictor variable is extended in the multivariate case to include several predictor variables, for example. Some multivariate techniques, like *Multiple Regression* and *Multivariate Analysis of Variance*, provides a way to perform in a single analysis what once took multiple univariate analyses to accomplish. Still, other techniques are designed exclusively to deal with multivariate issues, like *Factor Analysis* and *Discriminant Analysis*.

Multivariate techniques can be classified as: i) *Dependence Technique* or ii) *Interdependence Technique*. A dependence technique may be defined as one in which a variable

or set of variables is identified as the dependent variable to be predicted or explained by other variables (known as independent variables), *e.g.*, *Multiple Regression Analysis*. An interdependence technique is one in which no single variable or group of variables is defined as independent or dependent. Instead, all the variables are analyzed simultaneously to find an underlying structure to the entire set of variables or subjects., *e.g.*, *Exploratory Factor Analysis*.

Dependence Techniques can be categorized by two main characteristics: (1) the number of dependent variables, either a single dependent variable, several dependent variables, or several dependent/independent relationships, and (2) the type of measurement scale employed by the variables, either metric (quantitative/numerical) or nonmetric (qualitative/categorical).

- The appropriate techniques for an analysis that involves a single dependent metric variable are *Multiple Regression Analysis* or *Conjoint Analysis*, although the latter may treat the dependent variable as either nonmetric or metric, depending on the type of data collected;
- If the single dependent variable is nonmetric, then the appropriate techniques are *Multiple Discriminant Analysis* and *Logistic Regression*;
- When the analysis involves several dependent variables, four other techniques are suitable:
    i. *Look to the Independent Variables*, if the several dependent variables are metric;
    ii. *Multivariate Analysis of Variance*, if the independent variables are nonmetric;
    iii. *Canonical Correlation*, if the independent variables are metric; or
    iv. *Dummy Variable Coding* (*i.e.*, nonmetric data into metric data transformation) followed by *Canonical Analysis*, if the several dependent variables are nonmetric.
- The *Structural Equation Modeling* technique is appropriate when the analysis involves a set of dependent/independent variable relationships.

Interdependence Techniques cannot be categorized by having either dependent or independent variables. However, as with dependence techniques, the measurement properties should be considered.

- *Exploratory Factor Analysis* or *Confirmatory Factor Analysis*, generally considered to be metric techniques, are the appropriate ones when the structure of variables is to be analyzed;
- *Cluster Analysis* should be used if cases or respondents are to be grouped to represent structure;
- *Perceptual Mapping*, which has been developed for both metric and nonmetric approaches, is applicable if the interest is in the shape of objects;
- *Correspondence Analysis* is appropriate if the interdependencies of variables measured by nonmetric data are to be analyzed.

2.4.3 Multiple Regression Analysis

Hair *et al.* (HAIR et al., 2018) defined *Multiple Regression Analysis* as a statistical technique that can be used to analyze the relationship between a single dependent variable (known as the *criterion*) and several independent variables (known as the *predictors*). The independent variables, whose values are known, are exploited to explain or predict the single dependent value selected by the researcher.

This technique is termed *simple regression* when the problem involves a single independent variable or *multiple regression* when two or more independent variables are involved. Each independent variable is weighted to ensure maximal prediction from the independent variables collection. Variable weights denote the relative contribution of the independent variables to the overall prediction and ease interpretation as to the influence of each variable in making the prediction. Yet, correlation among the independent variables complicates the interpretative process. The set of weighted independent variables forms the *regression variate* (also known as the regression equation or regression model), the linear combination of the independent variables that best predict the dependent variable.

In the ***Simple Linear Regression Model***, the predictive power is determined by the association between the single dependent variable and the single independent variable, known as the *correlation coefficient*. Ranging from -1 to $+1$, the higher its value, the stronger the relationship and the greater the predictive accuracy. A $+1$ value indicates a perfect positive relationship, 0 appoints to no relation, and -1 designates a perfect negative or reverse relationship (*i.e.*, as one variable grows faster, the other variable grows slower). Moreover, the simple regression equation can be stated as $Y = b_0 + b_1 \cdot X_1 + \epsilon$,

where: Y is the *criterion variable*, the predicted or explained variable for any given value of the independent variable X_1; b_0 is the *intercept*, the constant term where the line defined by the regression equation crosses the Y axis (*i.e.*, the value of Y when $X_1 = 0$); b_1 is the *regression coefficient*, the numerical value of the parameter estimate directly associated with X_1 that represents the amount of change in Y for a one-unit change in X_1; X_1 is the *predictor variable*, expected to influence Y; and ϵ is the *residual* (prediction error of the sample data), the difference between the actual and predicted values of Y.

In the ***Multiple Linear Regression Model***, the goal is to expand upon the simple regression model by adding independent variables that have the highest additional predictive power. Adding more independent variables is based on trade-offs between increased predictive power versus overly complex and potentially misleading regression models. Unlike the simple model, the extent of the predictive power for any additional variable is often determined by its *collinearity* with other variables already defined in the regression equation. The collinearity expresses the relationship between two or more (*multicollinearity*) independent variables. Ranging from 0 to +1, the higher its value, the stronger the relationship. A +1 value indicates complete collinearity, and 0 a complete lack of collinearity. The prediction accuracy increases much more slowly as independent variables with high multicollinearity are added (due to the decrease of the unique variance explained by each independent variable). To maximize the prediction from a given number of independent variables, the researcher should look for independent variables that have low multicollinearity with the other independent variables and high correlations with the dependent variable. In an analogous way to the simple equation, the multiple regression equation can be stated as $Y = b_0 + \sum b_i \cdot X_i + \epsilon$, where i is the number of independent variables.

A *standardization* process is needed to directly compare the regression coefficients concerning their relative explanatory power of the dependent variable. It converts all independent variables to commons scale and variability (dispersion). A standardized unit of measurement allows for determining which variable has the most impact. The typical standardization procedure subtracts the variable *mean* from each observation's value to divide it by the *standard deviation* (the original variable is transformed into a new variable with a mean of 0 and a standard deviation of 1). When all the variables in a regression variate are standardized, the b_0 term (the intercept) assumes a value of 0, and the regression coefficients are known as *beta coefficients* (β). However, the beta coefficients should be used only to obtain the relative importance of the independent variables included in the equation and only for those variables with minimal multicollinearity.

Hair *et al.* (HAIR et al., 2018) elaborated a six-stage model-building process for the creation, estimation, interpretation, and validation of a regression analysis.

Stage 1: The objectives are specified, such as the appropriateness of the research problem, specification of a statistical relationship, and selection of the dependent and independent variables;

Stage 2: The researcher proceeds to design the regression analysis considering factors such as sample size, the need for variable transformations, and the nature of the relationship of the independent variables;

Stage 3: With the regression model formulated, the assumptions referred to in the regression analysis (*i.e.*, normality, linearity, homoscedasticity, and independence of the error terms) are first tested for the individual variables;

Stage 4: When all assumptions are met, the model is estimated. Diagnostic analyses are performed to ensure that the overall model meets the regression assumptions and that no observations have undue influence on the results (*e.g.*, coefficient of determination, adjusted coefficient of determination, standard error of the estimate, and statistical significance of regression coefficients);

Stage 5: The following steps are accomplished: interpretation of the regression variate, analysis of the relative importance of each independent variable in the prediction of the dependent measure, and evaluation of the multicollinearity and its effects;

Stage 6: The results are validated to ensure they represent the general population and can be applied to other contexts and studies.

2.4.4 Least Squares Estimation

As pointed out by Hair *et al.* (HAIR et al., 2018), the multiple regression analysis objective is most often achieved through the statistical rule of *Least Squares Estimation*. This procedure estimates the value of *regression coefficients* to minimize the total sum of the *squared residuals*, *i.e.*, reduce the discrepancies between observed data and their expected values. The estimator result is usually denoted by $\hat{\beta}$.

Van de Geer (VAN DE GEER, 2005) described the least squares method. In the regression problem, given the value of X (*predictor variables*), the best prediction of

Y (*criterion variable*) is the mean $f(X)$ of Y, where f is the regression function to be estimated from sampling n predictor variables and their data pairs responses, *i.e.*, $(x_1, y_1), \ldots, (x_n, y_n)$. Supposing f is known up to a finite number of regression coefficients $\beta = (\beta_1, \ldots, \beta_p)$, with $p \leq n$, then $f = f_\beta$. Therefore, β is estimated by the value b that gives the best fit to the data, *i.e.*, over all possible values, the one that minimizes **Equation 2.6**:

$$\hat{\beta} = \arg\min_b \sum_{i=1}^{n} (y_i - f_b(x_i))^2 \tag{2.6}$$

When f_β is a linear function of β, *i.e.*, $f_\beta(X) = X_1 \cdot \beta_1 + \ldots + X_p \cdot \beta_p$, it is more convenient to use a matrix notation. Let $y = (y_1, \ldots, y_n)$ be the vector of response variables and let $X = [x_1, \ldots, x_p]$ be the $n \times p$ data matrix of the n observations on the p variables, where x_j is the column vector containing the n observations on variable j, $j = 1, \ldots, n$. Thus, **Equation 2.7** express the squared distance between the vector y and the linear combination b of the columns of the matrix X.

$$\hat{\beta} = \arg\min_b \|y - X \cdot b\|^2 \tag{2.7}$$

Chen & Plemmons (CHEN; PLEMMONS, 2010) highlighted that estimation is not always straightforward since, for many real-world problems, the underlying parameters represent quantities that can take on only non-negative values. The *Non-Negative Least Squares* (NNLS) is a *constrained least-squares problem* type where the regression coefficients are not allowed to become negative. So, the previous problem must be modified to include nonnegativity constraints, as formulated in **Equation 2.8**.

$$\begin{aligned} &\hat{\beta} = \arg\min_b \|y - X \cdot b\|^2 \\ &\text{subject to } b \geq 0 \end{aligned} \tag{2.8}$$

Some tools implement the NNLS algorithm proposed by Lawson & Hanson (LAWSON; HANSON, 1987), an active set method that solves the *Karush-Kuhn-Tucker* (KKT) conditions for the non-negative least squares problem. For instance, *MathWorks MATLAB's lsqnonneg* (MATHWORKS, 2022) and *SciPy optimize's nnls* (SCIPY, 2022).

2.4.5 Measures of Predictive Performance for Regression Models

Hair *et al.* (HAIR et al., 2018) explained that predictive quality is always crucial to ensure the validity of the independent variables, as its measurement is used to assess the significance of their predictive power. In all instances, whether or not the researcher intends to interpret the coefficients of the variate, the regression analysis must achieve acceptable levels of predictive performance to justify its application.

The measures used in the regression models take into account how close the predicted values are to the actual values and report the prediction errors, unlike other evaluation measures such as *Accuracy Classification Score*, which assess the exact correspondence of predicted values with the actual values.

Regarding suitable metrics of predictive accuracy for regression models, the *Coefficient of Determination* (R^2 Score) is the most commonly used, as indicated by Hair *et al.* (HAIR et al., 2018), along with *Mean Square Error* (MSE), *Root Mean Square Error* (RMSE), and *Mean Absolute Error* (MAE). The last three are among the most popular metrics used by the researchers over a timeline of 25 years, as reported by Botchkarev (BOTCHKAREV, 2019).

Let n be the size of the data set (number of samples), A the set of actual (observed) values, $\bar{A}$ the mean (arithmetic average) of A, P the set of predicted values, $d_j = A_j - \bar{A}$ the deviation (error) between the j-th value of A and $\bar{A}$, and $e_j = A_j - P_j$ the difference (residual) between the j-th values of A and P, with $j \in \{1, \ldots, n\}$.

The **Coefficient of Determination** (R^2 Score) metric represents the amount of variance in the dependent variable that has been explained by the independent variable(s) in the model, the combined effects of the entire variate (one or more independent variables plus the intercept) in predicting the dependent variable. It indicates the goodness of fit, a measure of how well-unseen samples are likely to be predicted by the model using the proportion of explained variance that ranges from 0.0 (no prediction) to 1.0 (perfect prediction). Since the variance metric is dataset-dependent, R^2 may not be compared meaningfully across different datasets. Botchkarev (BOTCHKAREV, 2019) defined R^2 as follows (**Equation 2.9**):

$$R^2 = 1 - \frac{\sum_{j=1}^{n} {e_j}^2}{\sum_{j=1}^{n} {d_j}^2} \tag{2.9}$$

If the regression model is well applied and estimated, then the researcher can assume that the higher the value of R^2, the greater the explanatory power of the regression

equation. Furthermore, it allows for better prediction of the dependent variable. Pal & Gauri (PAL; GAURI, 2010) hinted as a thumb rule, *i.e.*, based on experience and common sense, that if the value of R^2 is more than 0.90, then the fitted model will be considered adequate.

The **Mean Squared Error** (MSE) is a risk metric corresponding to the expected value of the squared error loss (L2-norm loss function). Derived from the square of Euclidean distance, it is always a positive value that decreases as the error approaches zero. A zero value would indicate a perfect fit to the data, although it is rarely achieved in practice. Botchkarev (BOTCHKAREV, 2019) defined MSE as follows (**Equation 2.10**):

$$MSE = \frac{\sum_{j=1}^{n} {e_j}^2}{n} \tag{2.10}$$

The **Root Mean Squared Error** (RMSE) metric represents the square root of the average of squared errors. The effect of each error is proportional to the size of the squared error. Thus, it is sensitive to outliers, *i.e.*, to the observations that have a substantial difference between the actual value for the dependent variable and the predicted value. Being the square root of MSE, it is also always a non-negative value. Botchkarev (BOTCHKAREV, 2019) defined $RMSE$ as follows (**Equation 2.11**):

$$RMSE = \sqrt{\frac{\sum_{j=1}^{n} {e_j}^2}{n}} \tag{2.11}$$

The **Mean Absolute Error** (MAE) is a risk metric corresponding to the expected value of the absolute error loss (L1-norm loss function). It measures the average magnitude of the errors in a set of predictions. Derived from the Manhattan or taxicab distance, it is a positive value that decreases as the error approaches zero. Unlike $RMSE$, each error influences MAE in direct proportion to the absolute value of the error. Botchkarev (BOTCHKAREV, 2019) defined MAE as follows (**Equation 2.12**):

$$MAE = \frac{\sum_{j=1}^{n} |e_j|}{n} \tag{2.12}$$

Scikit-learn (SCIKIT-LEARN, 2022) is an open-source machine learning library for the Python programming language that provides, among many other features, a range of metrics and scoring for evaluating the quality of a model's predictions. In particular, the *regression metrics submodule* is of great use for this study since it implements many loss, score, and utility functions to measure regression performance.

3 Biological Sequences Comparison

3.1 Problem Definition

Biological Sequences play a crucial role in genetics and bioinformatics research. They are essential for describing *deoxyribonucleic acid* (DNA), *ribonucleic acid* (RNA), and *protein* biomolecules. Also are among the basic entities studied in molecular and computational biology.

3.1.1 Major Types and Basic Characteristics of Biomolecules

Biomolecules, also known as biological molecules, are any of the numerous substances produced by cells and living organisms. They have a wide range of sizes, structures, and functions. The four major types of biomolecules are (ENCYCLOPÆDIA BRITANNICA, 2022a):

i. *Carbohydrates*;

ii. *Lipids*;

iii. *Nucleic Acids*;

iv. *Proteins*.

Among the biomolecule types, *nucleic acids* (namely *DNA* and *RNA*) have the unique function of storing an organism's genetic code — the sequence of *nucleotides* that determines the *amino acid sequence* of proteins, which are fundamental to life on Earth.

The *DNA* carries the genetic information for the development and functioning of an organism. It is composed of two linked *strands* that wind around each other (a shape known as a double helix that resembles a twisted ladder). Each strand has a backbone formed by alternating *sugar deoxyribose* and *phosphate* groups. Attached to each sugar is

one of the following four *nitrogenous bases*: *Adenine* (A), *Cytosine* (C), *Guanine* (G), or *Thymine* (T). The two strands are connected by chemical bonds between the bases, such that *adenine* base pairs with *thymine* and *cytosine* base pairs with *guanine*. The sequence of the bases along DNA's backbone encodes biological information, *e.g.*, the instructions for generating a protein or RNA molecule (NHGRI, 2022a).

The *RNA* is a nucleic acid molecule found in all living cells with structural similarities to DNA. However, unlike DNA, RNA is most often single-stranded. It is composed of a backbone formed by alternating *phosphate* groups and *sugar ribose* (rather than the deoxyribose found in DNA). Attached to each sugar is one of the following four *nitrogenous bases*: *Adenine* (A), *Uracil* (U), *Cytosine* (C), or *Guanine* (G). During the DNA transcription step, the *uracil* base pairs with *adenine* (replacing *thymine*), while the *cytosine* base pairs with *guanine*. Three main types of RNA are involved in protein synthesis: *messenger RNA* (mRNA), *transfer RNA* (tRNA), and *ribosomal RNA* (rRNA). Certain *viruses* use RNA as their genomic material (NHGRI, 2022c), *e.g.*, *SARS-CoV-2 coronavirus* (the virus responsible for *COVID-19* disease).

3.1.2 Nucleotide Sequence

DNA and RNA molecules are long chains of nucleotides. Each nucleotide consists of a sugar molecule (either ribose in RNA or deoxyribose in DNA) attached to a phosphate group and a nitrogenous base (NHGRI, 2022b).

The *nucleotide sequence* is the most fundamental level of knowledge of a gene or genome. It contains the instructions for building an organism (*i.e.*, the recessive and dominant genes' data), and no understanding of the genetic function or variation could be complete without obtaining this information (ENCYCLOPÆDIA BRITANNICA, 2022b). *Automatic sequencing machines* (DNA sequencers) are generally used to determine the nucleotide sequence of an organism. The resulting sequence data are stored in a computer as text files and then shared in publicly accessible repositories (ENCYCLOPÆDIA BRITANNICA, 2022c), such as *National Center for Biotechnology Information* (NCBI, 2022a) and *Global Initiative on Sharing All Influenza Data* (GISAID, 2022).

In particular, viruses with an RNA genome, *e.g.*, *SARS-CoV-2 coronavirus*, are isolated (determined) from the genetic material of cells in the infected host (WU et al., 2020). Therefore, viral genome sequences are sorted by the host in public repositories.

3.1.3 Biological Sequences Analysis

Durbin *et al.* (DURBIN et al., 1998) emphasized that the most basic sequence analysis task is determining if two sequences are related. It is usually done by aligning the sequences (or parts of them) and then deciding whether that alignment is more likely to have occurred because they are related or just by chance — the *Pairwise Alignment*.

The key issues are:

i. *What sorts of alignment should be considered*. Sequences accumulate insertions, deletions as well as substitutions due to mutations. Before the similarity evaluation of two sequences, one typically begins by finding a plausible alignment between them;

ii. *The scoring system used to rank alignments*. The basic mutational processes considered are substitutions, which change residues in a sequence, and insertions and deletions, which add or remove residues (referred to as gaps, which usually have an associated penalty). The total score we assign to an alignment will be a sum of terms for each aligned pair of residues, plus each gap term;

iii. *The algorithm used to find optimal (or good) scoring alignments*. When both sequences have the same length, there is only one possible global alignment. It becomes more complicated when allowing gaps or searching for local alignments between subsequences. The algorithm for finding optimal alignments given an additive alignment score is known as dynamic programming, which is guaranteed to find the optimal scoring alignment or set of alignments. However, they are not the fastest available sequence alignment methods. In many cases, speed is an issue. The heuristic alignment algorithms search as small a fraction as possible of the cells in the dynamic programming matrix while still looking at all the high-scoring alignments;

iv. *The statistical methods used to evaluate the significance of an alignment score*. From the assessment, determine if it is a biologically meaningful alignment or just the best alignment between two entirely unrelated sequences.

The DNA sequences are generally compared with sequence alignment algorithms, which produce a textual file with highlighted matching characters. However, when highly similar sequences are compared, the biologists are most interested in their differences, which may indicate mutations, *e.g.*, *SARS-CoV-2 coronavirus* and its variants.

Sometimes the conventional sequence alignment is unwanted during the analysis. For instance, Lau *et al.* (LAU et al., 2021) developed a rapid computational approach to identify the highly conserved regions of *SARS-CoV-2 coronavirus* sequences to index mutations across thousands of viral genomes. Their approach avoids the inefficient behavior of the sequence alignment applied on a scale of thousands of genome sequences and facilitates viral pangenome studies.

3.2 Spark Implementation

This section describes the implementation details of the *Diff Sequences Spark*[1], a simple Spark application developed to illustrate the problem of provisioning resources in the Cloud to execute MapReduce-like Spark applications, aiming at cost optimization. Rather than proposing a novel pairwise sequence alignment tool, our implementation identifies, comparing position-by-position, all the mismatching characters occurrences, given the input sequences and the desired characters interval region for each.

3.2.1 Optimizing and Estimating the Number of Sequences Comparisons

A particular requirement for our application is how to optimize and estimate the total number of sequence comparisons to be processed. Let's denote it as $\boldsymbol{D_a}$.

Often the biological sequences are compared through the pairwise comparison approach, which forms subsets of cardinality 2, *i.e.*, two elements (sequences) per subset. The order of sequences in each formed pair is not relevant: given two sequences seq_1 and seq_2, it is equivalent to comparing seq_1 with seq_2 or seq_2 with seq_1, which reduces by half the number of comparisons produced. Furthermore, no subset may contain repeated sequences to be compared, *e.g.*, $\{\text{seq}_1, \text{seq}_1\}$. Under these restrictions, it is possible to apply the closed-form function of the *binomial coefficient* to obtain D_a. Let n be the total number of distinct elements and k be the subsets' cardinality. Graham *et al.* (GRAHAM et al., 1994) expressed the binomial coefficient function, denoted as $\binom{n}{k}$, in the multiplicative form as follows:

$$\binom{n}{k} = \frac{(n)_k}{k!} = \frac{n \cdot (n-1) \ldots (n-(k-1))}{k \cdot (k-1) \ldots (1)}, \qquad \text{where } n, k \in \mathbb{Z} \text{ and } 0 \leq k \leq n \tag{3.1}$$

Equation 3.1 gets simplified for the pairwise comparison case. In particular, $k = 2$

[1] https://github.com/alan-lira/diff-sequences-spark

and at least two sequences are required for the biological sequences comparison problem ($n \geq 2$). Thus, D_a can be estimated as follows:

$$D_a = \binom{n}{2} = \frac{n \cdot (n-1)}{2}, \qquad \text{where } n \in \mathbb{Z} \text{ and } n \geq 2 \tag{3.2}$$

Consider an implementation of the pairwise comparison approach, named ***DIFF*$_1$**. Let n be the number of distinct biological sequences. On each comparison procedure, two Spark distributed data structures are used to store the sequences: one sequence per data structure. Minding **Equation 3.2**, a great value of n implies a massive number of all-pairs comparisons, increasing the total runtime. For instance, $n = 1000$ sequences to be compared generates $D_a = 499\,500$ pairwise comparisons.

A simple and reasonable way to reduce the number of comparison procedures is discussed next. Let n be the number of distinct biological sequences. Consider an approach where a given sequence seq_i is compared against all its later sequences $\text{seq}_{i+1}, \ldots, \text{seq}_n$, $\forall i \in n$: seq_1 with $\{\text{seq}_2, \ldots, \text{seq}_n\}$, …, seq_{n-1} with $\{\text{seq}_n\}$. On each comparison procedure, two Spark distributed data structures are used to store the sequences: the first for the seq_i and the second for the $\text{seq}_{i+1}, \ldots, \text{seq}_n$ sequences. Notice that the last sequence seq_n does not have a later sequence and therefore cannot be compared to an empty subset. It is trivial to verify that, through this approach, D_a can be estimated as follows:

$$D_a = n - 1, \qquad \text{where } n \in \mathbb{Z} \text{ and } n \geq 2 \tag{3.3}$$

For instance, $n = 1000$ sequences to be compared generates $D_a = 999$ comparisons. Although the number of comparison procedures (**Equation 3.3**) becomes substantially reduced when compared to the *DIFF*$_1$ approach, it is unfeasible in practice for a large n because the data loading and its immediate processing in the second data structure may lead to:

- In the worst case, an *OutOfMemoryError* exception caused by insufficient space in the JVM heap, which abruptly terminates the application execution; or
- A non-negligible delay experienced by Spark's Scheduler for scheduling each comparison task. A maximum serialized task size of 100 Kilobytes (KB) is recommended for any Stage of the application's DAG (APACHE, 2022f). However, this information is only known during the application execution, when Spark prints the serialized size of each task in the Master (APACHE, 2022d).

To overcome these Spark limitations, consider the following enhanced approach named ***DIFF*$_{opt}$**. Let n be the number of distinct biological sequences. A given sequence seq_i is still compared against all its later sequences $\text{seq}_{i+1}, \ldots, \text{seq}_n$, $\forall i \in n$. Once again, the last sequence seq_n does not have a later sequence and therefore cannot be compared to an empty subset. On each comparison procedure, two Spark distributed data structures are used to store the biological sequences: the first for the seq_i and the second for a subset of the $\text{seq}_{i+1}, \ldots, \text{seq}_n$ sequences. Notice that no subset may contain repeated later sequences of seq_i (*i.e.*, already compared to seq_i). Let l_i be the number of later sequences of seq_i and let max_S be the maximum number of biological sequences that may be stored in the second data structure, with $1 \leq max_S < n$. Thus, $\lceil \frac{l_i}{max_S} \rceil$ subsets of the seq_i's later sequences are formed, and $[1, \ldots, l_i]$ comparisons are generated for every seq_i, depending on the value of max_S. Therefore, D_a can be estimated as follows:

$$D_a = \begin{cases} \left\lceil \frac{n}{max_S} \cdot \left((n-1) - \left(\frac{n - max_S}{2} \right) \right) \right\rceil + \epsilon, & \text{if } 1 \leq max_S < \frac{n}{2} \\ 2 \cdot \left((n-1) - \left(\frac{max_S}{2} \right) \right), & \text{if } \frac{n}{2} \leq max_S < n \end{cases} \tag{3.4}$$

$$\text{where } n, max_S \in \mathbb{Z} \text{ and } n \geq 2$$

The piecewise-defined function (**Function 3.4**) established for the $DIFF_{opt}$ approach was built from the empirical analysis of a small set of n and max_S values combinations (*cf.* **Appendix A**), and it is defined by two equations. The first one is valid when max_S belongs to the $[1, \frac{n}{2}[$ subdomain, and the second is valid when max_S belongs to the $[\frac{n}{2}, n[$ subdomain. Notice there may be an approximation error ϵ associated with the first equation, depending on n and max_S values. Also, notice **Equation 3.2** is a particular case of $DIFF_{opt}$ when $max_S = 1$, and **Equation 3.3** is a case of $DIFF_{opt}$ when $max_S = n - 1$.

This improved approach potentially reduces the number of sequences comparison procedures and avoids the execution issues related to running out of memory or delayed scheduling of tasks when using Spark. For instance, $n = 1000$ sequences to be compared using $max_S = 100$ sequences per data structure generates $D_a = 5490$ comparisons (first equation), and $n = 1000$ sequences to be compared using $max_S = 500$ sequences per data structure generates $D_a = 1498$ comparisons (second equation). Nevertheless, it is still essential to fine-adjust the value of max_S accordingly to the type of the biological sequences. Since different types have diverse quantities of nucleotide letters, the amount of data to be loaded and processed on each comparison varies from each case.

3.2.2 Diff Sequences Spark Application's Execution Flow

For each Spark application submitted to a Spark cluster, unique SparkContext and Executors JVM processes are created. However, Spark is typically optimized for massive data processing rather than low latency jobs. Thus, an overhead is associated with JVM process initialization. If each biological sequences comparison is submitted to the cluster individually as micro applications, the required time to spawn its processes may exceed the comparison processing time, mainly for small-sized biological sequences. Our implementation uses a single SparkContext (and its associated Executors) during the entire execution, which is responsible for iterating over all the comparison jobs left, in a serial flow fashion, to avoid multiple processes initializations overhead.

As pointed out by Zaharia *et al.* (ZAHARIA et al., 2012b), the MapReduce model can be expressed in the Spark framework through the following transformation operations:

- *flatMap*, for the *Map* phase; and
- *groupByKey* or *reduceByKey*, for the *Reduce* phase.

The *flatMap* transformation flattens the input Spark data structure (DataFrame or RDD) by applying a user-implemented function to each element. This transformation outputs a new Spark data structure (of the same type) that can have the same number of data elements or even more. It is the main difference between using *flatMap* and *map* transformations on Spark since the use of *map* always returns the same number of elements contained in the input data structure.

The *groupByKey* transformation iterates through each key-value pair $\langle K, V \rangle$ of the input data structure and groups the V_i values of a given K_i key into a single data element. This transformation outputs a new data structure (of the same type) with K key-value pairs, where K is the amount of K_i keys in the input data structure. This wide transformation triggers a data shuffle operation (*i.e.*, data rearranging between partitions).

The *reduceByKey* transformation merges the V_i values of a given K_i key of the input data structure into a single data element by applying a user-implemented reduce function. This transformation also outputs a new data structure (of the same type) with K key-value pairs, where K is the amount of K_i keys in the input data structure. The main difference between *reduceByKey* and *groupByKey* is that *reduceByKey* performs the merging locally on each mapper before sending the results to a reducer (data shuffling),

similarly to a *Combiner* in the MapReduce model. Thus, the *reduceByKey* may obtain better performance with larger datasets when compared to *groupByKey*.

Our Spark application implements, with eventual equivalent adaptations, the use of the *flatMap* and *reduceByKey* transformations, such that:

* The ***Map phase*** is composed of the *Read* and *Create* routines; and
* The ***Reduce phase*** is composed of the *Compare* and *Write* routines.

Figure 4 illustrates the execution flow of the *Diff Sequences Spark* application, whose routines are detailed below.

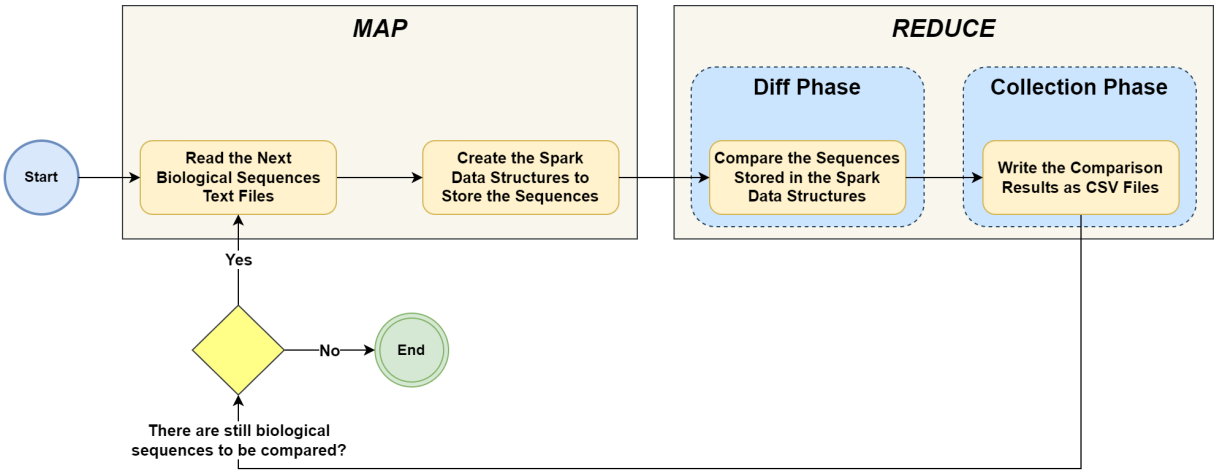


Figure 4: *Diff Sequences Spark* application's execution flow.

I. The **Read** routine, currently performed locally by the *Master* node, *i.e.*, without the aid of Spark transformations, is responsible for reading the *FASTA*-formatted file (*i.e.*, a text-based format for representing the nucleotide sequences using single-letter codes) of each biological sequence to be compared in the current iteration. It splits the chain of nucleotide letters of a given biological sequence into key-value pairs $\langle K, V \rangle$, where each K_i key stores the index position i of the i-th letter and each V_i value stores the respective i-th letter, and loads it in a local in-memory data structure. Two data structures are generated per iteration.

   Notice that for any comparison iteration, a biological sequence may have a lesser data size (*i.e.*, amount of letters) than the other(s). Thus, for some index positions, there might be an absence of data (letter), which requires proper treatment during the comparison (*Compare* routine).

$DIFF_1$ approach

The $DIFF_1$ approach allows two biological sequences to be compared per iteration. In this case, each data structure is filled with the data of one of them, *i.e.*, one sequence per structure.

$DIFF_{opt}$ approach

The $DIFF_{opt}$ approach allows two or more biological sequences to be compared per iteration. In this case, the second data structure is filled with multiple sequences' data. So, each K_i key of the second data structure is associated with a list of V_j values, where each V_j is the i-th letter of the j-th biological sequence.

II. The **Create** routine is responsible for creating the Spark data structures that will store the data generated in the *Read* routine.

Using *RDD*s

When choosing *RDD* as Spark's data structure, it is enough to use the *parallelize* transformation. It takes each data from the local data structures in the *Master* (Driver Program) and splits it into logically distributed partitions. With the key-value pairs data organization, this transformation outputs a particular type of *RDD*, named *PairedRDD*. Two *PairedRDD*s, denoted as rdd_1 and rdd_2, are built in this routine, one per data structure. However, they must be united into a single *PairedRDD*, denoted as rdd_u, before advancing to the *Compare* routine. This step is reached through the *union* transformation, which, in practice, appends the rdd_2 to the end of the rdd_1.

Using *DataFrame*s

When choosing *DataFrame* as Spark's data structure, we first specify its schema, *i.e.*, the column names and data types that define its structure, the *StructType*. The first column of the schema is a *LongType* for storing the index position, and the subsequent column(s) is(are) *StringType* for storing the nucleotide letter(s) corresponding to the index. The *createDataFrame* transformation takes each data from the local data structures in the *Master* (Driver Program), fills each DataFrame's column accordingly to the previously defined schema, and splits it into logically distributed partitions. Two *DataFrame*s denoted df_1 and df_2 are built in this routine, one per data structure.

III. The **Compare** (or **Diff Phase**) routine is responsible for comparing the Spark data structures' elements built in the *Create* routine.

Using *RDD*s

The rdd_u is processed through the *reduceByKey* transformation, which applies a *reduce function* over all its elements. The reduce function takes two elements with the same key (*i.e.*, index position), one originally belonging to the rdd_1 and another originally belonging to the rdd_2, and compare their values (*i.e.*, nucleotide letters).

If the $DIFF_1$ approach is used, then the reduce function provides three output situations:

i. The nucleotide letter from one of the elements is missing (*null*). In this case, the index position (key) is marked as incomparable;

ii. The nucleotide letters from both elements are identical. In this case, the index position (key) is marked as rejected;

iii. The nucleotide letters from both elements are different. In this case, the index position (key) is marked as accepted.

If the $DIFF_{opt}$ approach is used, then the reduce function iterates through each value of the element with multiple sequences stored (*i.e.*, the rdd_2 element) and provides three other output situations:

i. The nucleotide letter from one of the elements is missing (*null*). In this case, the index position (key) is marked as incomparable;

ii. The nucleotide letters from both elements are all identical. In this case, the index position (key) is marked as rejected;

iii. At least one nucleotide letter of both elements is different. In this case, every identical letter found in the multiple sequences element is replaced by the '=' symbol, and the index position (key) is marked as accepted.

After the *reduceByKey* transformation execution, the *filter* transformation is used to discard all the elements from the rdd_u whose keys were not marked as accepted, generating as output the rdd_r.

Using *DataFrame*s

The df_1 and df_2 are processed through the *join* transformation. Similar to the *SQL* clause, it combines the elements from two *DataFrame*s, given a *condition* and a *type* of join.

The *join condition* specifies how the rows from df_1 will be combined with the rows of df_2. In particular, the first column's value of df_1 and df_2 must be identical (*i.e.*, index position), and the subsequent column's value (*i.e.*, nucleotide letter) on both df_1 and df_2 must be different in the $DIFF_1$ approach. On the other hand, at least one of the subsequent columns of df_2 must be different from the corresponding column of df_1 in the $DIFF_{opt}$ approach.

The *join type* specified was the *full outer join*, which returns the combined values from both *DataFrame*s, appending *null* values on the side (*i.e.*, *DataFrame*'s column) that does not have a match. In particular, when the biological sequences have a different amount of nucleotide letters, they are incomparable for a given row.

After the *join* transformation execution, the *filter* transformation is used to discard all the rows whose index position's columns are filled with *null* values. Additionally, the *drop* transformation is used to remove the index column duplicated due to the join operation (originally belonging to df_2), generating as output the df_r.

If the $DIFF_{opt}$ approach is used, then the *withColumn* transformation is executed to find and replace with the '=' symbol the columns that have identical values (*i.e.*, nucleotide letters) for each row of df_r.

IV. The **Write** (or **Collection Phase**) routine is responsible for consolidating the comparisons results obtained in the *Compare* routine. Two approaches are available: *Distributed Write* (*DW*) and *Merged Write* (*MW*).

_DW_ approach

In the *DW* approach, each partition of Spark's data structure, which holds a chunk of the comparison results, is written by the Spark's Executor designated to that partition. Consequently, the output of each comparison iteration is stored in p files, where p is the number of partitions defined for Spark's data structure.

_MW_ approach

In the *MW* approach, the *coalesce* transformation is used in Spark's data structure, merging its data into a single partition. Consequently, the output of each comparison iteration is stored in a single file, written by a designated Executor.

Using _RDD_s

Two transformations are executed in the rdd_r: *sortByKey* and *map*. The *sortByKey* is used to sort the elements of rdd_r in ascending order. The *map* is used to format

each element of rdd_r with its representation similar to the comma-separated values (CSV) pattern, making the output data easier to read.

Finally, the *saveAsTextFile* action is executed, which consolidates the transformations performed in the previous routines. It persists the rdd_r's formatted data, as a text file, into a non-volatile disk storage destination previously specified.

Using *DataFrames*

The *sort* transformation is executed in the df_r to sort its rows by ascending order of the first column (*i.e.*, index position).

Finally, the *write.csv* action is executed, which consolidates the transformations performed in the previous routines. It persists the df_r's data, as a CSV file, into a non-volatile disk storage destination previously specified.

3.2.3 Illustrative Examples of the Comparisons Results

Consider four micro sequences to be compared using the *Diff Sequences Spark* application ($\boldsymbol{n = 4}$): $\{\text{seq}_1, \text{seq}_2, \text{seq}_3, \text{seq}_4\}$.

- The $\mathbf{seq_1}$ contains the following chain of nucleotide letters: $\langle\mathbf{AAAAA}\rangle$;
- The $\mathbf{seq_2}$ contains the following chain of nucleotide letters: $\langle\mathbf{ACGAA}\rangle$;
- The $\mathbf{seq_3}$ contains the following chain of nucleotide letters: $\langle\mathbf{GAGCA}\rangle$;
- The $\mathbf{seq_4}$ contains the following chain of nucleotide letters: $\langle\mathbf{ATAAAT}\rangle$.

In the first example, the $\boldsymbol{DIFF_1 + MW}$ approach is used. In this case, **Equation 3.2** determines the number of sequences comparisons ($\boldsymbol{D_a = 6}$). **Figure 5** illustrates the comparisons results obtained in this scenario.

In the second example, the $\boldsymbol{DIFF_{opt} + MW}$ approach is used. Consider a maximum of three sequences to be stored in the second data structure of any comparison procedure ($\boldsymbol{max_S = 3}$). In this case, **Function 3.4** determines the number of sequences comparisons ($\boldsymbol{D_a = 3}$). **Figure 6** illustrates the comparisons results obtained in this scenario.

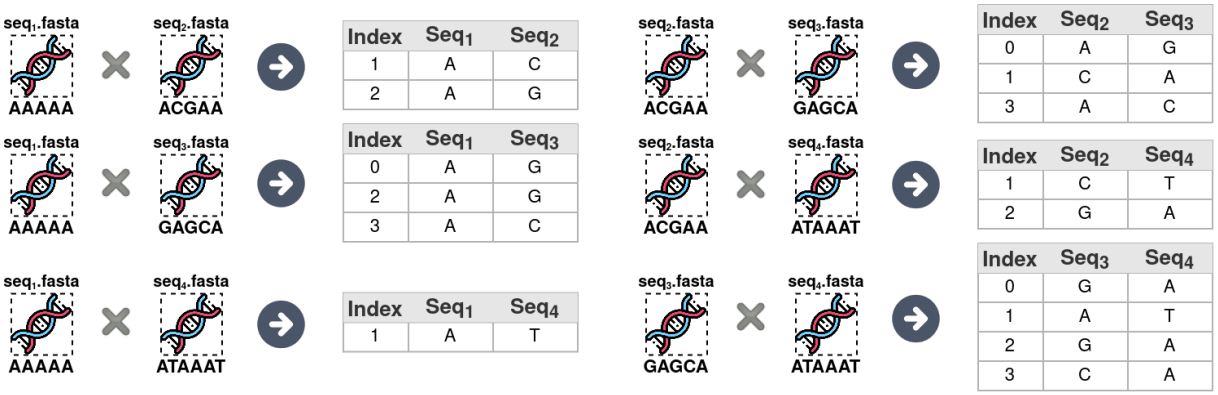


Figure 5: Comparisons results obtained with the $DIFF_1 + MW$ approach.

Figure 6: Comparisons results obtained with the $DIFF_{opt} + MW$ approach.

3.2.4 Application-Level Optimization Proposals to Reduce the Runtime

This subsection presents some proposals for reducing the *Diff Sequences Spark* application's runtime, considering the challenge of correctly setting the number of the partitions, the serial execution bottleneck of the application itself, and the default behaviors of Spark's cluster for scheduling jobs and for sharing resources across the active jobs.

I. **Regarding the Spark data structure's number of partitions**

Spark automatically sets the number of partitions accordingly to the type of task. For some map-like operations (*e.g.*, *read.textFile*), it considers each input file size, and for some reduce-like operations (*e.g.*, *groupByKey* and *reduceByKey*), it takes the largest parent RDD's number of partitions. Yet, this automatic setup may not be beneficial for all cases. If Spark's data structure is split into too many partitions, the tasks scheduling overhead may overcome their actual processing time. In contrast, when few partitions are defined, less concurrency is achieved.

(a) **First solution: general recommendations of Spark**

Following the general recommendations of Spark (APACHE, 2022e), one possible improvement is to set 2-3 partitions (tasks) per CPU core in the cluster. In this first solution, each Spark's data structure is repartitioned to three times the number of cores available through the *repartition* transformation.

(b) **Second solution: damping coefficient**

Let E_c be the total number of Executors cores (*i.e.*, the Spark's *default parallelism*), E_m the size in GiB of Executors memory, m_i the number of map tasks per comparison iteration, M the total number of map tasks, r_i the number of reduce tasks per comparison iteration, and R the total number of reduce tasks.

Since each comparison iteration is composed of two Spark's data structures (*i.e.*, either *RDD* or *DataFrame*), and each data structure is auto split into E_c partitions, then $m_i = 2 \cdot E_c$. Thus, $M = m_i \cdot D_a = 2 \cdot E_c \cdot D_a$.

The r_i value depends on which Spark's data structure is in use. Spark's default values are: $r_i = 2 \cdot E_c$ for two united *RDD*s, or $r_i = 200$ for two joined *DataFrame*s (Spark's *default shuffle partitions*). By employing the *DW* approach, $R = r_i \cdot D_a$. Using the *MW* approach, the resulting data structure of the *Compare* routine (*i.e.*, rdd_r or df_r) coalesces into a single partition. Therefore, $R = 1 \cdot D_a = D_a$.

Let $K = \{k \in \mathbb{N}^* \mid k \setminus E_c\}$ be the set of all the divisors of E_c. We can specify a damping coefficient $k_i \in K$ that acts over E_c for each Spark's data structure, which allows us to adjust the number of partitions to a value lesser than the one produced by the *default parallelism*. Consequently, $M = 2 \cdot \frac{E_c}{k_i} \cdot D_a$. Note that when $k_i = 1$, the *default parallelism* is applied. Furthermore, an optimal damping coefficient $k_{opt} \in K$ reduces the application runtime the most compared to each $k_i \in K \setminus \{k_{opt}\}$.

II. **Regarding the serial execution flow of *Diff Sequences Spark***

The *Read* and *Create* routines combined are equivalent to reading (*read.textFile*), mapping (*map*), and flatting (*flatMap*) the biological sequences data since each sequence is flattened in i index positions and parallelized to a Spark data structure. However, they are not performed exclusively through Spark transformations, which adds an execution bottleneck during the data loading of each comparison. The biological sequences files are centralized and sequentially loaded in the Master node (Driver program), so the Executors get idle, awaiting data receiving when the

execution moves from one comparison iteration to the next.

(a) **First solution: Spark built-in transformations on Map phase**

Through the use of the *read.textFile*, *map*, and *flatMap* transformations, to decentralize the biological sequences files off the Master node, achieving data loading parallelism per comparison iteration. The difficulty lies in implementing the current Master node's loading logic, so each Executor knows which files should be loaded. This solution has not been implemented yet for the *Diff Sequences Spark* application.

(b) **Second solution: multi-threading pool**

Spark supports concurrent job execution in the cluster, opposite the default behavior that runs one job per time in a *First In, First Out* mode (*FIFO*). Multiple jobs (Spark actions) within each Spark application may concurrently run if submitted by different threads (APACHE, 2022c). Consider the *Producer-Consumer* pattern, where the Producer threads are responsible for producing the Spark data structures of the current iteration (Map phase), and the Consumer threads are responsible for sending the already built Spark data structures for processing (Reduce phase). The sum of Producer and Consumer threads should be bounded by the number of CPU cores in the Master. This strategy ensures that Spark jobs are always ready to be processed since they are stored in a fixed-size thread-safe shared queue.

III. **Regarding the Spark's sharing of resources across the active jobs**

In addition to the *Producer-Consumer* solution, it is possible to set the *FAIR* scheduling mode using one or more fair pools for a balanced sharing of resources from the active Executors across the active jobs. Through a single resources pool, every job has the same priority for the available resources' usage in the cluster.

Appendix B presents the evaluation of the above-proposed application-level optimizations, allowing the selection of the most optimized configuration available to execute the *Diff Sequences Spark* application, considering the $DIFF_1 + MW$ approach. All experiments in this work specified in the following chapters employ this configuration.

3.3 SARS-CoV-2 Sequences Comparisons: A Study Case

The sudden emergence of the severe respiratory disease known as *COVID-19*, caused by a novel severe acute respiratory syndrome, the *SARS-CoV-2* (CSC, 2020), has recently become a public health emergency at a pandemic outbreak level, threatening human beings worldwide.

3.3.1 Classification and Basic Characteristics of Coronaviruses

Coronaviruses (*CoV*s) are a family of viruses that cause illness in humans and animals. They belong to the subfamily *Coronavirinae*, family *Coronaviridae*, and order *Nidovirales* and are classified into four genera:

i. *Alphacoronavirus* (α-CoVs);

ii. *Betacoronavirus* (β-CoVs);

iii. *Gammacoronavirus* (γ-CoVs); and

iv. *Deltacoronavirus* (δ-CoVs).

Currently, there are seven types of CoVs known to cause infections in humans: *229E* (α-CoV), *NL63* (α-CoV), *OC43* (β-CoV), *HKU1* (β-CoV), *SARS-CoV* (β-CoV), *MERS-CoV* (β-CoV), and *SARS-CoV-2* (β-CoV). The first four types typically cause mild clinical symptoms, such as rhinorrhea, cough, and fever. However, the last three types can affect respiratory, gastrointestinal, hepatic, and nervous systems and may cause respiratory failure, multiple organ dysfunction, and even death (BAHRAMI; FERNS, 2020).

CoVs are enveloped viruses with a single-strand, positive-sense RNA genome (*i.e.*, the viral genome can be directly translated into viral proteins) ranging from 26 to 32 kilobases in length. Since each kilobase contains a thousand base pairs of RNA, the CoVs hold the longest known genome for an RNA virus (WEISS; NAVAS-MARTIN, 2005).

In particular, the SARS-CoV-2's pathology level (*i.e.*, the propensity to cause disease to the infected hosts) is higher than the previously related coronaviruses, identified as Severe acute respiratory syndrome, the SARS-CoV (DROSTEN et al., 2003), and Middle East respiratory syndrome, the MERS-CoV (ZAKI et al., 2012). Genome sequence analysis of SARS-CoV-2 revealed a close resemblance to them (NAQVIA et al., 2020): it shares about 82% sequence identity with SARS-CoV and MERS-CoV and 90% or more sequence

identity for essential enzymes and structural proteins, such as spike (S), envelope (E), membrane (M), and nucleocapsid (N). Besides, all three are zoonotically transmitted (*i.e.*, transmission between species, from animals to humans, and vice versa) and spread among humans through close contact. The primary reproduction number R_0 (*i.e.*, the transmissibility of infectious agents) of SARS-CoV-2 was about 2.6, which significantly contributed to the exponential growth of infection cases (HELLEWELL et al., 2020).

3.3.2 NCBI's Viral Sequences Data Repository

The *NCBI Virus* (HATCHER et al., 2017) is a value-added viral sequence data resource hosted by the *National Center for Biotechnology Information* (NCBI). It includes modules for many viral groups, such as *influenza virus*, *Dengue virus*, *West Nile virus*, *Ebolavirus*, *MERS coronavirus*, *Rotavirus A*, *Zika virus*, and most recently, *SARS-CoV-2 coronavirus*. The users select publicly available sequences based on standardized gene, protein, and metadata terms. After selection, a suite of tools supports user-directed activities, such as retrieving the nucleotide or protein sequences data (as FASTA-formatted files) for research. However, when the user wants to retrieve multiple sequences at once, the NCBI Virus system merges them into a single downloadable file. Thus, an auxiliary tool may be required to split them. For that purpose, we developed the *Fasta Sequences File Splitter*[2], a command-line tool to split one multiple FASTA-formatted sequences file into individual FASTA-formatted sequences files.

Presently, around 5.77 million nucleotide records (DNA sequences), with humans as the infected host, are available for the SARS-CoV-2 virus at *NCBI Virus* (NCBI, 2022b). Due to the massive amount of data, the analysis is often done by geographical region and period, comparing a subset of sequences to each other (*i.e.*, the all-against-all matching).

3.3.3 SARS-CoV-2 Nucleotide Sequences Comparisons on AWS EC2

To illustrate the problem of optimizing computational costs for MapReduce-like Spark applications on the cloud, we have selected at the *NCBI Virus* a subset of *540 SARS-CoV-2 full-length nucleotide sequences*[3], with humans as the infected host, from the South America region. It is input data of the *Diff Sequences Spark* application, which will be executed using different launching characteristics for several Spark clusters deployed in Amazon's AWS EC2 cloud service.

[2] https://github.com/alan-lira/fasta-splitter
[3] https://doi.org/10.17605/OSF.IO/TDEPK

4 Related Work

Regarding performance improvement of Big Data applications on the Cloud at a minimal cost, the literature provides works that either tackle the resource provisioning problem, the application scheduling considering monetary costs and deadlines, or the provision of a formalization of the problem. **Table 1** summarizes the related work, emphasizing their main contribution concerning the proposed framework, the objective, the applications evaluated, and the environment considered.

Table 1: Related work.

Reference	Framework	Main Contributions	Applications and Workloads	Experimental Environments
(CHEN et al., 2014)	*CRESP*	• Efficient resource provisioning for MapReduce Hadoop applications in public clouds; • Execution time optimization, subject to budget constraint; • Execution cost optimization, subject to deadline constraint.	◇ WordCount; ◇ Sort; ◇ PageRank; ◇ TableJoin.	∗ In-house Cluster; ∗ Amazon AWS EC2.
(ZHAO et al., 2015)	*SparkSW*	• Implementation of the Smith-Waterman algorithm on Apache Spark.	◇ Protein sequences of UniProt Reference Clusters (UniRef100).	∗ In-house Cluster.
(XU, B. et al., 2017a)	*DSA*	• Implementation of scalable distributed sequence alignment system on Apache Spark; • Use of data parallel strategy based on SIMD instruction to parallelize the algorithm in each core of worker node; • Use of Alluxio as primary storage.	◇ Protein sequences of UniProt Reference Clusters (UniRef100).	∗ In-house Cluster.

(Continued on the next page)

Table 1: Related work. (cont.)

Reference	Framework	Main Contributions	Applications and Workloads	Experimental Environments
(XU, B. et al., 2017b)	*CloudSW*	• Scalable implementation of the Smith-Waterman algorithm in two modes: – alignment scores (ASM) and – alignment traceback (ATM); • Implementation of an API tool service.	◇ Protein sequences of UniProt Reference Clusters (UniRef100).	∗ In-house Cluster; ∗ Alibaba Aliyun E-MapReduce.
(YAN et al., 2016)	*TR-Spark*	• Execution cost prediction; • Transient instances stability prediction.	◇ TPC-DS Benchmark (SQL-like Jobs); ◇ Bing API; ◇ Cortana Session; ◇ PageRank.	∗ Home-built TR-Spark Simulator; ∗ Microsoft Azure Batch.
(XU, F. et al., 2018)	*iSpot*	• Transient instances stability prediction; • Spark application performance modelling; • Analytical checkpointing mechanism.	◇ WordCount, Grep, and Sort (BigDataBench); ◇ Alternating Least Squares (ALS); ◇ Frequent Pattern Growth (FP-Growth); ◇ PageRank (Spark MLlib).	∗ Amazon AWS EC2; ∗ Google GCE.
(XU, Y. et al., 2020)	*Dagon*	• DAG scheduling; • Resources use maximization; • Cache-aware management.	◇ LinearRegression, LogisticRegression, and DecisionTree (SparkBench); ◇ K-Means and TriangleCount (SparkBench); ◇ ConnectedComponent and PregelOperation (SparkBench).	∗ In-house Cluster.
(ISLAM et al., 2020)	*ILP* and *BFD*	• Scheduling algorithms that reduces the execution costs; • Prioritization of jobs based on their given deadlines.	◇ WordCount, Sort, and PageRank (BigDataBench).	∗ Nectar Research Cloud.

(Continued on the next page)

Table 1: Related work. (cont.)

Reference	Framework	Main Contributions	Applications and Workloads	Experimental Environments
(ISLAM et al., 2021)	*SLA Scheduler*	• Service-level agreement (SLA) Spark job scheduler in a hybrid cloud.	◇ WordCount, Sort, and PageRank (BigDataBench).	∗ Home-built Event-Based Simulator; ∗ Hybrid Cluster (Nectar Research Cloud virtual machines from two different regions).

Chen *et al.* (CHEN et al., 2014) proposed *CRESP* (an acronym for Cloud RESource Provisioning), a resource provisioning optimizer method for running MapReduce applications in public clouds. The work focus on the performance analysis of the following critical variables: the logic and time complexity of the Reduce function for a specific application, the input data size, and the number of available Mappers (map slots) and Reducers (reduce slots) on the cluster. With these variables in hand, Chen *et al.* (CHEN et al., 2014) provided a time cost model, expressed as a weighted linear combination of a set of non-linear functions. The weights can be learned through small-scale experiments of execution time measurements for a particular application applied to the non-negative linear regression problem. Chen *et al.* (CHEN et al., 2014) formulated two optimization problems: minimizing time within a monetary budget or cost within a deadline constraint. The experiments were conducted on in-house and AWS EC2 clusters to validate the cost model. Low error rates were obtained from the model prediction considering *WordCount*, *Sort*, *PageRank*, and *TableJoin* applications. Results showed the benefits of their approach on the execution time reduction or monetary cost compared to randomly selected cluster configurations.

The following studies can be highlighted as aiming at works in the literature related to Spark applications. *SparkSW*, as remarked by Zhao *et al.* (ZHAO et al., 2015), is the first Spark-based implementation of the Smith-Waterman algorithm that provides scalability and load-balancing for biological sequence pairwise alignment in a distributed environment. By employing a Spark MapReduce Model on HDFS, *SparkSW* implements: data pre-processing, the main calculations of Smith-Waterman performed by the map tasks, and a summary operation achieved by the reduce tasks. An experimental evaluation was carried out on a small in-house cluster consisting of four nodes, each with 188 GB of RAM and 32 processing cores, to align a significant number of sequences. However, the

size of the local cluster was not expressive regarding scalability.

DSA, presented by Xu *et al.* (XU, B. et al., 2017a), leverages a data-parallel strategy based on SIMD instructions to parallelize the algorithm on each core associated with a Worker. It employs the memory-based distributed file system *Alluxio* as primary storage to speed up I/O performance and reduce network traffic. A performance comparison with *SparkSW* was executed considering an in-house cluster, showing that *DSA* achieved up to 201x speedup over *SparkSW*.

CloudSW, introduced by Xu *et al.* (XU, B. et al., 2017b), also leverages Spark and SIMD instructions to accelerate SW algorithm, supports alignment scores and trace-backs, and provides services to run in the cloud. Experimental results showed that *CloudSW* achieves up to 3.29 times speedup over *DSA* and 621 times over *SparkSW*. Note that the Intel SIMD instructions used in *DSA* and *CloudSW* are not portable across microarchitectures, *e.g.*, they cannot be executed in ARM processors.

Works have been exploring the spot instances usage to reduce operating costs. *TR-Spark*, issued by Yan *et al.* (YAN et al., 2016), allows the execution of Spark applications on transient resources based on resource stability, data size reduction aware scheduling, and lineage-aware checkpointing process. *TR-Spark* specifies background tasks on nodes not fully utilized for their primary assigned tasks. These re-computation costs are minimized by backing up intermediate results according to the resources instability level, re-computation cost, and data lineage. *TR-Spark* also prioritizes those tasks that output the least amount of data. Also, a proactive checkpointing policy is devised by saving data blocks before the virtual instances fail, defined following a probabilistic approach. Although Yan *et al.* (YAN et al., 2016) plan to minimize checkpointing overheads, revocation time is hard to predict accurately. Albeit the paper has shown promising performance results, their work focus on the stability prediction of the VMs.

Also tackling transient instances in the cloud, Xu *et al.* (XU, F. et al., 2018) presented *iSpot*. It evaluates the stability degree during a job execution based on the Long Short-Term Memory (LSTM) method, an artificial recurrent neural network capable of learning order dependence in sequence prediction problems. Furthermore, *iSpot* models the Spark applications' performance via automatic job profiling. It also provides an analytical performance checkpointing mechanism, considering the Spark performance model, data checkpointing, and restoration overheads. Although the results show a decrease in the financial costs, their analyses rely only on spot instances.

Islam *et al.* (ISLAM et al., 2020) proposed a scheduling framework developed on

top of *Apache Mesos* that provides two online/dynamic algorithms: the *Integer Linear Programming* model and the *Best Fit Decreasing* heuristic. Their output enhances job performance and minimizes the allocation cost of heterogeneous VMs in the cloud. The main focus is to deploy a Spark cluster through cost-effective Executors placement among Workers nodes for any job in the cluster, maximizing resource utilization while prioritizing jobs based on their given deadlines. This work deployed a prototype of *SLA-Scheduler*, established by Islam *et al.* (ISLAM et al., 2021). The benefits of their approach were evaluated using *WordCount*, *Sort*, and *PageRank* applications from the *BigDataBench* benchmark, published by Wang *et al.* (WANG et al., 2014), against two scheduling techniques, Spark's *FIFO* and *Morpheus*, the latter disclosed by Jyothi *et al.* (JYOTHI et al., 2016). It obtained resource cost reduction of up to 34% and, for mixed and network-bound jobs, a completion time reduction of up to 14%.

Extending their work, Islam *et al.* (ISLAM et al., 2021) tackled the problem of executing Spark applications on a hybrid cloud computing cluster. In this version, the scheduler analyses only jobs that meet their deadlines to make a cost-effective executor placement decision. It is reached by profiling each job to estimate the completion time, used as input for the proper job scheduling in the VMs. Reinforcing the use of *SLA-Scheduler*, two algorithms were proposed and included in the framework: a modified version of the *First Fit* and a *Greedy* approach that iteratively places all the Executors of a job in the most cost-optimal position. Initially, extensive simulation-based experiments were performed to analyze their proposals against baseline algorithms. Later, Islam *et al.* (ISLAM et al., 2021) also developed a prototype on top of Apache Mesos cluster manager and executed experiments in a real cluster using a mixed workload of WordCount, Sort, and PageRank applications from *BigDataBench*. Indeed, their proposed scheduling algorithms reduced the costs by up to 20%, but only compared to other strategies that did not consider the assignment of Executors to a hybrid Spark cluster.

Xu *et al.* (XU, Y. et al., 2020) proposed *Dagon*, a middleware that maximizes throughput by exploiting task resource demands for the stages of the DAG. It dynamically reconfigures resources to run the tasks through the following mechanisms: *DAG-aware task assignment*, *sensitivity-aware delay scheduling*, and *priority-aware cache management*. The first reduces resource fragmentation, the second prevents executors from wasting resources due to unnecessary waiting for high-locality tasks, and the last makes cache eviction and prefetches decisions based on the stage priority determined by the first mechanism. *Dagon* was implemented on the Spark framework through modifications of the *DAGScheduler*, *TaskScheduler*, *BlockManagerMaster*, and *BlockManager* modules. Com-

parisons were carried out between *Dagon* and *Graphene*, a dependency-aware scheduler implemented by Grandl *et al.* (GRANDL et al., 2016), and between *Dagon* and *MRD*, a reference-distance-based cache management scheme implemented by Perez *et al.* (PEREZ et al., 2018). *Dagon* reduced job completion time by up to 42% and increased the CPU utilization by up to 46%, considering workloads from the *SparkBench* benchmark introduced by Li *et al.* (LI et al., 2017). In this work, we note that it is technically possible to influence task scheduling in Spark.

5 Building the Diff Sequences Spark Application's Runtime Prediction Model

5.1 Review of the *CRESP*'s MapReduce Time Cost Model

Chen *et al.* (CHEN et al., 2014) proposed *CRESP* (an acronym for *Cloud RESource Provisioning*), a resource provisioning optimizer method to execute MapReduce Hadoop applications in public clouds. Understanding their MapReduce time cost model allowed us to propose a runtime prediction model for the *Diff Sequences Spark* application.

5.1.1 Initial Assumptions

When *CRESP* was proposed, Chen *et al.* (CHEN et al., 2014) didn't use the *YARN* manager (also known as *MapReduce 2.0* or *MRv2*). So, at the time, the following characteristics of the *Hadoop* framework (APACHE, 2022a) were considered for their modeling:

- The *JobTracker* daemon in the *Master* node is responsible for scheduling and assigning computational tasks to the *Worker* nodes;
- The *TaskTracker* daemon in a *Worker* node is composed of the *map slots* (*Mappers*), which run *map tasks*, and the *reduce slots* (*Reducers*), which run *reduce tasks*;
- According to the cluster capacity, a *Worker* can only accommodate a fixed number of slots. Typically, a multi-core *Worker* allocates one slot per CPU core, either map or reduce;
- An application (*Job*) is driven by the number of input data splits, which are dependent upon the block size defined on *HDFS*; and
- Each *map task* handles one chunk of the input data.

Chen *et al.* (CHEN et al., 2014) assumed there are m *map slots* (*Mappers*) and r *reduce slots* (*Reducers*) in total, distributed over the *Worker* nodes of a cluster. Besides, if there are M chunks of data, then M *map tasks*, in total, will be scheduled and assigned to the m slots. In the ideal case, m *map tasks* run simultaneously (a *Map phase* round). If $M > m$, then $\lceil \frac{M}{m} \rceil$ *Map phase* rounds are required. R *reduce tasks* are to be executed in total, a setting defined manually by the user or automatically by Hadoop. If $R > r$, more than one *Reduce phase* round is required. However, unlike the *map tasks*, there is no data size restriction for *reduce tasks*. So, usually, $R = r$.

5.1.2 Cost of the Map and Reduce Tasks

Chen *et al.* (CHEN et al., 2014) analyzed the MapReduce execution process to determine the cost of the *map* and *reduce tasks*.

A *Map Task* is divided into three main components, which are sequentially executed: *Read*, *Map*, and *Sort/Partition*.

i. The *Read* component reads, either locally or remotely, a block b of data from the disk. The average cost, denoted as $i(b)$, is determined by the input data size of b;

ii. The *Map* component, a user-defined function, maps block b into a list of key-value pairs $\langle k, v \rangle$, whose size, denoted as $o_m(b)$, might vary depending on the block b's data. Its cost, denoted as $f(b)$, is also determined by the input data size of b;

iii. The *Sort/Partition* component sorts by key the key-value pairs list $o_m(b)$ into R partitions for the R reduce tasks. The cost for sorting and partitioning $o_m(b)$ is denoted as $s(o_m(b), R)$.

In practice, *map tasks* in the same *Map phase* round may not finish simultaneously due to the system configuration, disk I/O rate variation, network traffic performance variation, and data distribution. Thus, an error component, denoted as ϵ_m, is included in the *map task* cost modeling. The overall cost of a *map task*, denoted as Φ_m, is the sum of the costs of the above components. It is defined as follows (**Equation 5.1**):

$$\Phi_m = i(b) + f(b) + s(o_m(b), R) + \epsilon_m \tag{5.1}$$

Notice the $i(b)$ and $f(b)$ components are related to the data size of block b and the complexity of the map function, independently of m and M parameters. Also, notice the

$s(o_m(b), R)$ component is slightly linear to R. Thus, Φ_m is defined in terms of m, M, r, and R as follows (**Equation 5.2**):

$$\Phi_m(m, M, r, R) = \mu_1 + \mu_2 \cdot R + \epsilon_m \tag{5.2}$$

Where, μ_1, μ_2, and ϵ_m are application-specific constants.

A *Reduce Task* is divided into four main components, which are sequentially executed: *Shuffle*, *MergeSort*, *Reduce*, and *WriteResult*. Assuming that the k keys of the *Map phase* are equally distributed to the R *reduce tasks*, the total amount of data for each *reduce task* is $b_R = M \cdot o_m(b) \cdot \frac{k}{R}$. Furthermore:

i. The *Shuffle* component pulls the partition's corresponding data (*i.e.*, the key-values pairs outputted from the *Map phase*). Most of the time, it is overlapped with the *Map phase*. Usually, only the *Map phase*'s last round contributes to this component cost, denoted as $c(b_R)$. Thus, $c(b_R) \approx m \cdot o_m(b) \cdot \frac{k}{R}$;

ii. The *MergeSort* component merges by key the partition's pulled data through multiple rounds. Assuming a buffer size B, then $\lceil \log_B(M) \rceil$ merge rounds are required. Its cost is denoted as $ms(b_R)$. Thus, $ms(b_R) \propto b_R \cdot \lceil \log_B(M) \rceil$;

iii. The *Reduce* component, a user-defined function, reduces (*i.e.*, process) the partition's merged data with an application-specific complexity, denoted as $g(b_R)$. It is assumed that the output data size, denoted as $o_r(b_R)$, is often less than b_R;

iv. The *WriteResult* component writes back the result to multiple nodes. Its cost, denoted as $w_r(o_r(b_R))$, is determined by $o_r(b_R)$.

Due to disk I/O rate variation and network traffic performance variation, the costs for the *Shuffle* and *WriteResult* components may vary. Thus, an error component, denoted as ϵ_r, is included to *reduce task* cost modeling. The overall cost of a *reduce task*, denoted as Φ_r, is the sum of the costs of the above components. It is defined as follows (**Equation 5.3**):

$$\Phi_r = c(b_R) + ms(b_R) + g(b_R) + w_r(o_r(b_R)) + \epsilon_r \tag{5.3}$$

Notice the $c(b_R)$ component is linear to b_R, the $ms(b_R)$ component is proportional to b_R, and the $w_r(o_r(b_R))$ component is linear to $o_r(b_R)$. Keeping the relevant components, Φ_r is defined in terms of m, M, r, and R as follows (**Equation 5.4**):

$$\Phi_r(m, M, r, R) = \lambda_1 \cdot \left(\frac{m}{R}\right) + \lambda_2 \cdot \left(\frac{M \cdot \log_2(M)}{R}\right) + g\left(\frac{M}{R}\right) + \lambda_3 \cdot \left(\frac{M}{R}\right) + \epsilon_r \tag{5.4}$$

Where, λ_1, λ_2, λ_3, and ϵ_r are application-specific constants.

5.1.3 *CRESP*'s Runtime Prediction Model

Chen *et al.* (CHEN et al., 2014) pointed out that the overall time complexity, denoted as T, depends on the number of *Map phase* and *Reduce phase* rounds. However, there is a cost, denoted as $\Theta(M,R)$, for Hadoop to manage and schedule the *map* and *reduce tasks*. According to them, $\Theta(M,R)$ is linear to M and R, such that (**Equation 5.5**):

$$\Theta(M,R) = \xi_1 \cdot M + \xi_2 \cdot R \tag{5.5}$$

Where, ξ_1, and ξ_2 are application-specific constants.

As such wise, the overall time cost T is represented as follows (**Equation 5.6**):

$$T = \left\lceil \frac{M}{m} \right\rceil \cdot \Phi_m + \left\lceil \frac{R}{r} \right\rceil \cdot \Phi_r + \Theta(M,R) \tag{5.6}$$

Chen *et al.* (CHEN et al., 2014) were more interested in the relationship between the dependent variable T and the following independent variables: the input data size $M \times b$, the user-defined number of *reduce tasks* R, the number of *map slots* m, and the number of *reduce slots* r. To this end, they assumed that:

- The number of *map tasks* M is a multiple of the number of *map slots* m, such that $M \geq m$ and $m \mid M$. Thus, the $\lceil \frac{M}{m} \rceil$ factor is simplified to $\frac{M}{m}$; and
- The number of *reduce tasks* R equals the number of *reduce slots* r. Thus, the $\lceil \frac{R}{r} \rceil$ factor is simplified to 1.

Plugging **Equations 5.2, 5.4**, and **5.5** into **Equation 5.6** and keeping only the variables M, R, and m in the cost model, we obtain (**Equation 5.7**):

$$
\begin{aligned}
T(M,m,R) = \mu_1 \cdot \left(\frac{M}{m}\right) + \mu_2 \cdot \left(\frac{M \cdot R}{m}\right) + \epsilon_m \; + \\
\lambda_1 \cdot \left(\frac{m}{R}\right) + \lambda_2 \cdot \left(\frac{M \cdot \log_2(M)}{R}\right) + g\left(\frac{M}{R}\right) + \lambda_3 \cdot \left(\frac{M}{R}\right) + \epsilon_r \; + \\
\xi_1 \cdot M + \xi_2 \cdot R
\end{aligned}
\tag{5.7}
$$

The complexity of *Reduce* component $g\left(\frac{M}{R}\right)$ has to be estimated with the specific application. However, if it is linear to the size of the input data, which is frequent according to Chen *et al.* (CHEN et al., 2014), then its contribution can be merged to the $\lambda_3 \cdot \left(\frac{M}{R}\right)$ part because $g\left(\frac{M}{R}\right) \sim \left(\frac{M}{R}\right)$. Also, the ϵ_m and ϵ_r noises can be combined into an overall noise variable denoted as ϵ. Thus, $T(M,m,R)$ is simplified to (**Equation 5.8**):

$$
\begin{aligned}
T(M,m,R) = \mu_1 \cdot \left(\frac{M}{m}\right) + \mu_2 \cdot \left(\frac{M \cdot R}{m}\right) \; + \\
\lambda_1 \cdot \left(\frac{m}{R}\right) + \lambda_2 \cdot \left(\frac{M \cdot \log_2(M)}{R}\right) + \lambda_3 \cdot \left(\frac{M}{R}\right) \; + \\
\xi_1 \cdot M + \xi_2 \cdot R + \epsilon
\end{aligned}
\tag{5.8}
$$

According to the transformed cost components meaning, the μ_1, μ_2, λ_1, λ_2, λ_3, ξ_1, and ξ_2 constants cannot assume negative values. Moreover, T is linear to its transformed cost components and non-linear to its variables M, m, and R. Therefore, it is defined as a weighted linear combination of a set of non-linear functions, such that (**Equation 5.9**):

$$
\begin{aligned}
T(M,m,R) = \beta_0 + \sum_{i=1}^{7} \beta_i \cdot x_i + \epsilon \\
\text{Where, } x_1 = \frac{M}{m}, x_2 = \frac{M \cdot R}{m}, x_3 = \frac{m}{R}, x_4 = \frac{M \cdot \log_2(M)}{R}, x_5 = \frac{M}{R}, x_6 = M, x_7 = R, \text{ and} \\
\beta_i \geq 0, \; i = 0 \ldots 7
\end{aligned}
\tag{5.9}
$$

Thus, $T(M,m,R)$ is expressed in the form of *multiple linear regression*. The *beta coefficients* β_i, which represent the contribution of each component x_i in the prediction model, vary from application to application. The β_0 (known as *intercept*) represents a constant cost of T when $x_i = 0$, $i = 1 \ldots 7$. Moreover, all the β_i can be derived from the *non-negative linear regression analysis* with measurements taken from experimental runs. The ϵ term represents the random error in measurements caused by unknown changes.

5.2 *Diff Sequences Spark*'s Runtime Prediction Model

Starting from the *CRESP* model proposed by Chen *et al.* (CHEN et al., 2014) and with the necessary adaptations to the *Spark* environment, we present below a runtime prediction model for the *Diff Sequences Spark* application.

5.2.1 Initial Assumptions

Regarding the *Spark* framework (APACHE, 2022b), at version v3.2.1, the following characteristics were considered for our modeling proposal:

- The *Driver* daemon in the *Master* node is responsible for scheduling and assigning computational tasks to the *Worker* nodes;
- The *Executor* daemon in a *Worker* node can run both *map* and *reduce* tasks, typically one task per CPU core;
- A *Worker* can accommodate zero, one, or multiple *Executors* according to the availability of its resources (CPU cores and RAM), which are split among them;
- The number of input data splits (partitions) can be programmatically adjusted at the application level, independently of the data loading function specified; and
- Each *map task* handles one chunk (partition) of the input data.

Consider there are E_c *Executors* cores, in total, distributed over the *Worker* nodes of a cluster. Besides, if there are M chunks of data, then M *map tasks*, in total, will be scheduled and assigned to the E_c cores. In the ideal case, E_c *map tasks* run simultaneously (a *Map phase* round). If $M > E_c$, then $\lceil \frac{M}{E_c} \rceil$ *Map phase* rounds are required. R *reduce tasks* are to be executed in total, a setting defined programmatically by the user or automatically by Spark. If $R > E_c$, then $\lceil \frac{R}{E_c} \rceil$ *Reduce phase* rounds are required.

5.2.2 Runtime Prediction Model

Through adapting **Equation 5.9**, and bearing in mind the assumptions described in the previous subsection, the *Diff Sequences Spark* application's runtime prediction model, denoted as T_{DSS}, is defined as follows (**Equation 5.10**):

$$T_{DSS}(M, R, E_c) = \beta_0 + \sum_{i=1}^{7} \beta_i \cdot x_i + \epsilon$$

$$\text{Where, } x_1 = \frac{M}{E_c}, x_2 = \frac{M \cdot R}{E_c}, x_3 = \frac{E_c}{R}, x_4 = \frac{M \cdot \log_2(M)}{R}, x_5 = \frac{M}{R}, x_6 = M, x_7 = R, \text{ and}$$
$$\beta_i \geq 0,\ i = 0 \ldots 7 \tag{5.10}$$

Thus, $T_{DSS}(M, R, E_c)$ is expressed in the form of *multiple linear regression*. We are more interested in the relationship between the dependent variable T and the following independent variables: the number of *map tasks* M, the number of *reduce tasks* R, and the number of *Executors cores* E_c. It is possible to predict the runtime cost T through learning the *beta coefficients* β_i from the *non-negative linear regression analysis* of experimental run measurements taken of the *Diff Sequences Spark* application.

5.3 Evaluation of the *Diff Sequences Spark*'s Prediction Model

Moving forward with experimental investigations about T and the influence of the M, R, and E_c variables, we evaluated the *Diff Sequences Spark*'s runtime prediction model using a specially designed dataset.

5.3.1 Experimental Environment and Application-Related Settings

We considered the following general settings to generate the runtime dataset:

I. **Regarding the experimental environment (Spark on AWS EC2)**

* AWS EC2 Memory Optimized Instances: the *z1d.6xlarge* instance (24 CPU cores and 192 GiB RAM) for the *Master* node, and the {*r5.large*, *r5.xlarge*, *r5.2xlarge*, *r5.4xlarge*, *r5.8xlarge*, *r5.12xlarge*, *r6i.large*, *r6i.xlarge*, *r6i.2xlarge*, *r6i.4xlarge*, *r6i.8xlarge*, *r6i.12xlarge*, *z1d.large*, *z1d.xlarge*, *z1d.2xlarge*, *z1d.3xlarge*, *z1d.6xlarge*, *z1d.12xlarge*} subset of instances for the *Worker* nodes. The memory optimized offers the best cost-benefit execution (NUNES et al., 2021);
* Spark Cluster: single *Master* node and {1, ..., 6} *Worker* nodes, with a single *Executor* per *Worker*;
* Spark's Scheduler: *FAIR* Standalone scheduler, which employs a single fair resource sharing pool.

II. **Regarding the *Diff Sequences Spark* application**

* Data Input: $n \in \{2, 4, 8, 16, 32, 64\}$ SARS-CoV-2 nucleotide sequences;
* Spark's Data Structure: *RDD*;
* Diff Phase: $DIFF_1$;
* Collection Phase: *MW*;
* Optimizations: damping coefficient $k_i = E_c$, and multi-threading pool composed of 12 producer and 12 consumer threads in the *Master*, with a maximum capacity of 300 ready-to-run comparison jobs in the shared queue.

Let ω_w be the Worker instance name and ι_w the number of Workers per experiment. Concerning each experimental measurement, since k_i is set to E_c, $M = 2 \cdot D_a$ *map tasks* are run in total (with D_a *Map phase* rounds required). Also, since the *MW* approach is used, $R = D_a$ *reduce tasks* are run in total (*i.e.*, one per sequences comparison job). Several experiments were executed (90 in total), with 70% (63) randomly selected for the *training set* and the remaining 30% (27) for the *testing set*.

5.3.2 Training the Prediction Model

Table 2 summarizes the T, M, R, and E_c values for the training set, which are considered in the learning step of the runtime prediction model. Through an *implementation*[1] that uses the *SciPy optimize's nnls* library (SCIPY, 2022) to solve the non-negative least squares curve fitting problem, and applying it to the training set experiments, we obtained $\hat{\beta} = \{\beta_0 = 0, \beta_1 = 0.011, \beta_2 = 0, \beta_3 = 0.083, \beta_4 = 0, \beta_5 = 5.034, \beta_6 = 0.055, \beta_7 = 0\}$, which are the weights (contributions) of the x_i components of **Equation 5.10**.

Table 2: *Diff Sequences Spark*'s prediction model training set.

Worker Node ω_w	ι_w	E_c	E_m	n	D_a	k_i	M	R	**Runtime T (Seconds)**
r5.large	1	2	14	8	28	2	56	28	27.82
	4	8	56	2	1	8	2	1	12.91
r5.xlarge	1	4	29	32	496	4	992	496	103.11

(Continued on the next page)

[1] https://github.com/alan-lira/crespark

Table 2: *Diff Sequences Spark*'s prediction model training set. (cont.)

Worker Node ω_w	ι_w	E_c	E_m	n	D_a	k_i	M	R	**Runtime T (Seconds)**
r5.xlarge	3	12	87	2	1	12	2	1	11.32
	5	20	145	16	120	20	240	120	23.69
	5	20	145	2	1	20	2	1	11.38
	2	8	58	32	496	8	992	496	73.53
r5.2xlarge	3	24	183	16	120	24	240	120	22.36
	3	24	183	64	2016	24	4032	2016	216.17
r5.4xlarge	6	96	738	8	28	96	56	28	14.13
	1	16	123	16	120	16	240	120	23.17
	2	32	246	8	28	32	56	28	14.43
	3	48	369	8	28	48	56	28	14.11
	4	64	492	8	28	64	56	28	14.07
r5.8xlarge	1	32	247	8	28	32	56	28	14.40
	2	64	494	64	2016	64	4032	2016	238.73
	6	192	1482	64	2016	192	4032	2016	235.59
	4	128	988	64	2016	128	4032	2016	237.52
r5.12xlarge	4	192	1488	32	496	192	992	496	59.92
	1	48	372	16	120	48	240	120	21.72
	3	144	1116	64	2016	144	4032	2016	233.22
r6i.large	4	8	56	64	2016	8	4032	2016	233.84
	5	10	70	8	28	10	56	28	15.34
	5	10	70	16	120	10	240	120	25.40
r6i.xlarge	3	12	87	64	2016	12	4032	2016	225.06
	4	16	116	8	28	16	56	28	14.55
	2	8	58	64	2016	8	4032	2016	240.27
r6i.2xlarge	1	8	60	8	28	8	56	28	14.25
	5	40	300	16	120	40	240	120	20.82
	1	8	60	32	496	8	992	496	66.18
	2	16	120	32	496	16	992	496	60.49
r6i.4xlarge	6	96	732	32	496	96	992	496	58.29
	4	64	488	64	2016	64	4032	2016	235.41

(Continued on the next page)

Table 2: *Diff Sequences Spark*'s prediction model training set. (cont.)

Worker Node		E_c	E_m	n	D_a	k_i	M	R	**Runtime T (Seconds)**
ω_w	ι_w								
r6i.4xlarge	5	80	610	16	120	80	240	120	20.50
	6	96	732	64	2016	96	4032	2016	235.75
r6i.8xlarge	4	128	984	64	2016	128	4032	2016	236.24
	3	96	738	32	496	96	992	496	58.27
	2	64	492	16	120	64	240	120	20.35
	5	160	1230	32	496	160	992	496	58.31
	2	64	492	64	2016	64	4032	2016	234.42
r6i.12xlarge	4	192	1480	64	2016	192	4032	2016	234.32
	3	144	1110	64	2016	144	4032	2016	237.69
z1d.large	6	12	84	16	120	12	240	120	25.81
	5	10	70	64	2016	10	4032	2016	226.08
	5	10	70	32	496	10	992	496	68.68
z1d.xlarge	3	12	87	16	120	12	240	120	23.83
	5	20	145	64	2016	20	4032	2016	221.79
	4	16	116	64	2016	16	4032	2016	228.71
z1d.2xlarge	2	16	122	16	120	16	240	120	22.65
	6	48	366	64	2016	48	4032	2016	235.06
	6	48	366	32	496	48	992	496	58.95
z1d.3xlarge	2	24	184	8	28	24	56	28	13.21
	4	48	368	8	28	48	56	28	12.75
	3	36	276	16	120	36	240	120	20.81
z1d.6xlarge	4	96	740	64	2016	96	4032	2016	231.75
	6	144	1110	64	2016	144	4032	2016	231.08
	4	96	740	32	496	96	992	496	57.91
	3	72	555	32	496	72	992	496	58.02
	6	144	1110	32	496	144	992	496	57.84
z1d.12xlarge	2	96	744	16	120	96	240	120	21.21
	4	192	1488	32	496	192	992	496	57.86
	3	144	1116	32	496	144	992	496	58.10
	4	192	1488	64	2016	192	4032	2016	233.49

5.3.3 Predicting with the Trained Model

We evaluated the prediction quality of our model by fixing the learned β_i coefficients values ($\hat{\beta}$) in **Equation 5.10** and by using the M, R, and E_c values from each experiment of the testing set to predict each respective T value, denoted as $T_{predicted}$. Then, each $T_{predicted}$ was compared to its actual T value, denoted as T_{actual}, to determine the prediction error for each experiment, denoted as Δ_T, where $\Delta_T = |T_{predicted} - T_{actual}|$ (*i.e.*, the absolute error). **Table 3** summarizes the T_{actual}, $T_{predicted}$, and Δ_T values of the testing set.

Finally, through an *implementation*[2] that uses the *Mean Absolute Error* (MAE), *Root Mean Squared Error* (RMSE), and *Coefficient of Determination* (R^2) regression metrics from the *Scikit-learn* library (SCIKIT-LEARN, 2022), the following regression scores were achieved: $MAE = 4.048$, $RMSE = 6.095$, and $R^2 = 0.994$. One should recall that the closest MAE and $RMSE$ are to zero, and the closest R^2 is to one, the most precise the regression model is. Thus, we obtained a very accurate model for the *Diff Sequences Spark* application.

Table 3: *Diff Sequences Spark*'s prediction model testing set.

Worker Node ω_w	ι_w	E_c	E_m	n	D_a	k_i	M	R	T_{actual} (Seconds)	$T_{predicted}$ (Seconds)	Δ_T
r5.large	5	10	70	16	120	10	240	120	28.81	23.47	5.34
	1	2	14	2	1	2	2	1	12.98	10.35	2.63
	4	8	56	8	28	8	56	28	18.57	13.23	5.34
r5.2xlarge	6	48	366	4	6	48	12	6	13.17	11.39	1.78
	3	24	183	8	28	24	56	28	14.64	13.23	1.41
	1	8	61	4	6	8	12	6	12.24	10.85	1.39
r5.8xlarge	2	64	494	16	120	64	240	120	21.58	23.29	1.71
r5.12xlarge	6	288	2232	64	2016	288	4032	2016	231.42	230.90	0.52
	1	48	372	32	496	48	992	496	60.17	64.59	4.42
r6i.large	5	10	70	64	2016	10	4032	2016	224.71	235.11	10.40

(Continued on the next page)

[2] https://github.com/alan-lira/crespark

Table 3: *Diff Sequences Spark*'s prediction model testing set. (cont.)

Worker Node ω_w	ι_w	E_c	E_m	n	D_a	k_i	M	R	T_{actual} **(Seconds)**	$T_{predicted}$ **(Seconds)**	Δ_T
r6i.large	5	10	70	32	496	10	992	496	67.29	65.44	1.85
r6i.xlarge	3	12	87	16	120	12	240	120	23.68	23.43	0.25
	6	24	174	16	120	24	240	120	22.50	23.33	0.83
r6i.2xlarge	4	32	240	64	2016	32	4032	2016	237.82	232.10	5.72
r6i.4xlarge	2	32	244	16	120	32	240	120	20.60	23.31	2.71
r6i.12xlarge	1	48	370	64	2016	48	4032	2016	231.84	231.65	0.19
	4	192	1480	32	496	192	992	496	58.21	64.45	6.24
	2	96	740	16	120	96	240	120	20.28	23.30	3.02
z1d.large	2	4	28	32	496	4	992	496	90.80	67.05	23.75
	2	4	28	8	28	4	56	28	19.33	13.30	6.03
z1d.xlarge	2	8	58	8	28	8	56	28	16.02	13.23	2.79
	5	20	145	32	496	20	992	496	61.68	64.90	3.22
z1d.2xlarge	4	32	244	8	28	32	56	28	13.99	13.25	0.74
	5	40	305	32	496	40	992	496	60.20	64.63	4.43
z1d.3xlarge	5	60	460	32	496	60	992	496	58.38	64.55	6.17
	3	36	276	32	496	36	992	496	59.07	64.66	5.59
z1d.12xlarge	1	48	372	64	2016	48	4032	2016	232.48	231.65	0.83

6 Optimizing the VMs Allocation on AWS EC2 for Diff Sequences Spark Application

6.1 Initial Assumptions

Consider the following characteristics for the computational environment on AWS EC2, the *Diff Sequences Spark* application, and the *Diff Sequences Spark*'s application runtime prediction model:

I. **Regarding the computational environment (Spark on AWS EC2)**

* The Spark cluster is composed of one or more *Master* nodes of the same VM instance type and one or more *Worker* nodes of the same VM instance type; and
* There is a maximum number of $vCPU$s (CPU cores) for all running VM instances a user can rent at a time ($vCPU$s quota constraint).

II. **Regarding the *Diff Sequences Spark* application**

* The M *map tasks* number is assumed to be known (estimated) before solving the optimization problem; and
* The R *reduce tasks* number is assumed to be known (estimated) before solving the optimization problem.

III. **Regarding the *Diff Sequences Spark*'s runtime prediction model**

* The β_i coefficients values, *i.e.*, $\hat{\beta} = \{\beta_0, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7\}$, are assumed to be known (learned) before solving the optimization problem.

Let ι_m be the number of *Master* nodes, where $\iota_m \geq 1$, let γ_m be the number of CPU cores per *Master* VM instance, let ι_w be the number of *Worker* nodes, where $\iota_w \geq 1$, and let γ_w be the number of CPU cores per *Worker* VM instance. The total number of CPU cores for the Spark cluster, already considering the *vCPU*s quota constraint, is defined as $T_c = (\iota_m \cdot \gamma_m) + (\iota_w \cdot \gamma_w)$. Therefore, the maximum number of CPU cores that can be acquired for the *Executors* is $E_{c_{max}} = T_c - (\iota_m \cdot \gamma_m)$. Moreover, the minimum number of CPU cores that can be acquired for the *Executors* is $E_{c_{min}} = \gamma_w$.

6.2 *Diff Sequences Spark*'s Monetary Cost Function

Let υ_m be the monetary cost per unit of time to rent each *Master* VM instance and υ_w be the monetary cost per unit of time to rent each *Worker* VM instance.

Consider the *Diff Sequences Spark* application's runtime cost function, denoted as T_{DSS} (**Equation 5.10**), with all its values known (user-provided) except for E_c. Bearing in mind the assumptions described in the previous section, the *Diff Sequences Spark* application's monetary cost function, denoted as C_{DSS}, is defined as follows (**Equation 6.1**):

$$C_{DSS}(E_c) = ((\iota_m \cdot \upsilon_m) + (\iota_w \cdot \upsilon_w)) \cdot T_{DSS}(E_c) \tag{6.1}$$

Where, $\iota_w = \left\lceil \frac{E_c}{\gamma_w} \right\rceil$, and M, R, ι_m, υ_m, γ_w, and υ_w are user-provided constants.

6.3 Optimization Problems for *Diff Sequences Spark*

Cloud users often have a challenging responsibility in picking the most appropriate VMs configuration to run their applications due to a wide variety of resources made available by the cloud providers. To propose a solution to overcome this challenge, we are interested in optimizing the VMs allocation to execute the *Diff Sequences Spark* application on AWS EC2, considering two conflicting minimization goals: runtime and financial cost. Furthermore, **Table 4** summarizes the notation defined for both objectives.

Table 4: *Diff Sequences Spark*'s optimization problems notations.

Notation	Description
T_{DSS}	Runtime cost function for *Diff Sequences Spark* (**Equation 5.10**).
C_{DSS}	Monetary cost function for *Diff Sequences Spark* (**Equation 6.1**).
β_i	Beta coefficients learned with *Multiple Regression Analysis* (user-provided).
M	Number of *map tasks* for *Diff Sequences Spark* (user-provided).
R	Number of *reduce tasks* for *Diff Sequences Spark* (user-provided).
ι_m	Number of *Master* nodes (user-provided).
υ_m	Monetary cost per unit of time to rent each *Master* VM instance (user-provided).
γ_w	Number of CPU cores per *Worker* VM instance (user-provided).
υ_w	Monetary cost per unit of time to rent each *Worker* VM instance (user-provided).
$E_{c_{min}}$	Minimum number of CPU cores that can be acquired for the *Executors* (user-provided).
$E_{c_{max}}$	Maximum number of CPU cores that can be acquired for the *Executors* (user-provided).
ϕ	Budget constraint to execute the *Diff Sequences Spark* application (user-provided).
τ	Time constraint to execute the *Diff Sequences Spark* application (user-provided).
$E_{c_{opt}}$	The optimal number of CPU cores for the *Executors* (optimization output).
ι_w	Number of *Worker* nodes to be acquired (derived from $E_{c_{opt}}$).

Notice that the number of *Worker* nodes to be acquired is derived from the $E_{c_{opt}}$ value, such that $\iota_w = \left\lceil \frac{E_{c_{opt}}}{\gamma_w} \right\rceil$.

6.3.1 Runtime Optimization Subject to Budget Constraint

Given a user-provided budget constraint denoted as ϕ (*i.e.*, the maximum monetary spend) to execute the *Diff Sequences Spark* application, the optimization problem that minimizes the runtime cost is defined as follows (**Equation 6.2**):

$$\begin{aligned} \textbf{Input:}\ & \beta_i, M, R, \iota_m, \upsilon_m, \gamma_w, \upsilon_w, E_{c_{min}}, E_{c_{max}}, \phi \\ \textbf{Output:}\ & E_{c_{opt}} \\ \textbf{Minimize}\ & T_{DSS} \\ \textbf{Subject to}\ & C_{DSS} \leq \phi \\ & E_{c_{opt}} \geq E_{c_{min}} \\ & E_{c_{opt}} \leq E_{c_{max}} \end{aligned} \tag{6.2}$$

6.3.2 Monetary Cost Optimization Subject to Deadline Constraint

Given a user-provided deadline constraint denoted as τ (*i.e.*, the maximum time spend) to execute the *Diff Sequences Spark* application, the optimization problem that minimizes the monetary cost is defined as follows (**Equation 6.3**):

$$\begin{aligned}
\textbf{Input:}\ & \beta_i, M, R, \iota_m, \upsilon_m, \gamma_w, \upsilon_w, E_{c_{min}}, E_{c_{max}}, \tau \\
\textbf{Output:}\ & E_{c_{opt}} \\
\textbf{Minimize}\ & C_{DSS} \\
\textbf{Subject to}\ & T_{DSS} \leq \tau \\
& E_{c_{opt}} \geq E_{c_{min}} \\
& E_{c_{opt}} \leq E_{c_{max}}
\end{aligned} \tag{6.3}$$

7 Experimental Results on AWS EC2

7.1 Evaluation of the *Diff Sequences Spark*'s Cost Optimizer

The experimental evaluation aims at the benefits of using the VMs allocation optimizer to execute the *Diff Sequences Spark* application on AWS EC2 in contrast to a random selection of VMs provided by an inexperienced cloud user. For that goal, we *implemented*[1] the optimization problems defined in the previous chapter using the *Gurobi Optimizer* solver (GUROBI, 2022) to obtain the optimal number of CPU cores for the *Executors* and, consequently, the appropriate Spark cluster size (*i.e.*, the number of *Master* and *Worker* nodes).

7.1.1 Experimental Environment and Application-Related Settings

We considered the general settings described in **Subsection 5.3.1** plus the following changes:

I. **Regarding the *Diff Sequences Spark* application**

   * Data Input: $n \in \{2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96\}$ SARS-CoV-2 nucleotide sequences.

II. **Regarding the *Diff Sequences Spark*'s optimization problems**

   * *Master* Nodes: $\iota_m = 1$ and $\upsilon_m = 2.232$ USD per hour of use;
   * Optimization Constraints: $E_{c_{min}} = \gamma_w$, $E_{c_{max}} = 10 \cdot E_{c_{min}}$, $\phi = 2.5$ USD, and $\tau = 0.2$ hours ($\equiv$ 12 minutes $\equiv$ 720 seconds).

[1] https://github.com/alan-lira/crespark

Suppose an inexperienced cloud user randomly selected the number of CPU cores for the *Executors*, denoted as $E_{c_{rnd}}$, such that $E_{c_{min}} \leq E_{c_{rnd}} \leq E_{c_{max}}$. Also, let T be the runtime cost in seconds and C be the monetary cost in USD to execute the *Diff Sequences Spark* application under several experimental configurations.

7.1.2 Minimizing the *Diff Sequences Spark* Application's Runtime Cost

Table 5 summarizes the results when the runtime minimization is aimed using the optimizer, with a maximum monetary spend $\phi = 2.5$ USD. Notice the optimal selection of parameters allowed an average runtime cost reduction of 3.6% compared to the scenarios where the user does not know the proper selection of parameters.

7.1.3 Minimizing the *Diff Sequences Spark* Application's Monetary Cost

Table 6 summarizes the results when the monetary cost minimization is aimed using the optimizer, with a maximum runtime spend $\tau = 0.2$ hours ($\equiv$ 12 minutes $\equiv$ 720 seconds). Notice the optimal selection of parameters allowed an average monetary cost reduction of 48.67% compared to the scenarios where the user does not know the proper selection of parameters.

7.2 *Diff Sequences Spark*'s Execution With Spot Instances

Since Spark provides a fault-tolerant environment, we also investigate the effect of using spot instances *Workers* in the runtime and monetary cost of the *Diff Sequences Spark* application, contemplating scenarios with either spot instances revocation or not. Notice the eventual loss of *Workers* shall not impact the execution completion as the *SparkContext* will be launched into the on-demand instance *Master* (*Driver program*).

7.2.1 Experimental Environment and Application-Related Settings

We considered the general settings described in **Subsection 5.3.1** plus the following changes:

Table 5: *Diff Sequences Spark*'s runtime cost optimization results.

n	OPTIMIZATION INPUTS								OPTIMAL SELECTION				RANDOM SELECTION				% CHANGE	
									Parameters		Results		Parameters		Results			
	γ_w	ω_w	v_w	ϕ	M	R	$E_{c_{min}}$	$E_{c_{max}}$	$E_{c_{opt}}$	ι_w	T (s)	C (USD)	$E_{c_{rnd}}$	ι_w	T (s)	C (USD)	T	C
2		r5.large	0.126	2.5	2	1	2	20	2	1	12.90	0.0084	16	8	13.72	0.0124	−6.03%	−31.61%
3	2	r6i.large	0.126	2.5	6	3	2	20	2	1	12.54	0.0082	4	2	12.53	0.0086	+0.13%	−4.95%
4		z1d.large	0.186	2.5	12	6	2	20	3	2	13.12	0.0095	6	3	14.17	0.0110	−7.36%	−13.54%
6		r5.xlarge	0.252	2.5	30	15	4	40	8	2	14.23	0.0108	20	5	15.18	0.0147	−6.25%	−26.55%
8	4	r6i.xlarge	0.252	2.5	56	28	4	40	14	4	14.58	0.0131	12	3	14.91	0.0124	−2.17%	+6.08%
12		z1d.xlarge	0.372	2.5	132	66	4	40	34	9	17.46	0.0271	24	6	17.25	0.0214	+1.24%	+26.54%
16		r5.2xlarge	0.504	2.5	240	120	8	80	61	8	22.22	0.0387	32	4	22.18	0.0262	+0.16%	+47.70%
24	8	r6i.2xlarge	0.504	2.5	552	276	8	80	80	10	35.95	0.0726	8	1	41.73	0.0317	−13.85%	+128.98%
32		z1d.2xlarge	0.744	2.5	992	496	8	80	80	10	58.68	0.1577	48	6	59.44	0.1106	−1.27%	+42.61%
48		z1d.3xlarge	1.116	2.5	2256	1128	12	120	120	10	126.79	0.4717	96	8	125.96	0.3905	+0.65%	+20.78%
64	12	z1d.3xlarge	1.116	2.5	4032	2016	12	120	120	10	239.72	0.8918	84	7	227.26	0.6341	+5.48%	+40.64%
96		z1d.3xlarge	1.116	2.5	9120	4560	12	120	120	10	551.83	2.0528	60	5	552.28	1.1984	−0.08%	+71.29%
2		r5.4xlarge	1.008	2.5	2	1	16	160	16	1	11.00	0.0099	48	3	11.98	0.0175	−8.12%	−43.36%
3	16	r6i.4xlarge	1.008	2.5	6	3	16	160	16	1	9.98	0.0090	64	4	10.39	0.0181	−3.92%	−50.31%
4		r6i.4xlarge	1.008	2.5	12	6	16	160	16	1	10.33	0.0093	80	5	14.65	0.0296	−29.50%	−68.59%
6		z1d.6xlarge	2.232	2.5	30	15	24	240	24	1	11.64	0.0144	48	2	11.95	0.0222	−2.63%	−35.09%
8	24	z1d.6xlarge	2.232	2.5	56	28	24	240	24	1	13.06	0.0162	144	6	14.61	0.0634	−10.56%	−74.44%
12		z1d.6xlarge	2.232	2.5	132	66	24	240	34	2	15.23	0.0283	96	4	15.66	0.0485	−2.72%	−41.63%
16		r5.8xlarge	2.016	2.5	240	120	32	320	61	2	21.61	0.0376	96	3	21.13	0.0486	+2.26%	−22.63%
24	32	r6i.8xlarge	2.016	2.5	552	276	32	320	141	5	35.58	0.1217	192	6	36.01	0.1433	−1.20%	−15.10%
32		r6i.8xlarge	2.016	2.5	992	496	32	320	254	8	58.28	0.2972	160	5	58.30	0.1994	−0.04%	+49.07%
48		r5.12xlarge	3.024	2.5	2256	1128	48	480	480	10	126.82	1.1440	96	2	126.72	0.2915	+0.08%	+292.48%
64	48	r6i.12xlarge	3.024	2.5	4032	2016	48	480	480	10	233.79	2.1088	336	7	224.03	1.4562	+4.36%	+44.82%
96		z1d.12xlarge	4.464	2.5	9120	4560	48	480	144	3	530.11	2.3007	384	8	557.90	5.8802	−4.98%	−60.87%

Table 6: *Diff Sequences Spark*'s monetary cost optimization results.

n	OPTIMIZATION INPUTS								OPTIMAL SELECTION				RANDOM SELECTION				% CHANGE	
									Parameters		Results		Parameters		Results			
	γ_w	ω_w	v_w	τ	M	R	$E_{c_{min}}$	$E_{c_{max}}$	$E_{c_{opt}}$	ι_w	T (s)	C (USD)	$E_{c_{rnd}}$	ι_w	T (s)	C (USD)	T	C
2		r5.large	0.126	0.2	2	1	2	20	2	1	12.99	0.0085	2	1	12.62	0.0083	+2.94%	+2.94%
3	2	r6i.large	0.126	0.2	6	3	2	20	2	1	12.36	0.0081	4	2	12.52	0.0086	−1.32%	−6.32%
4		z1d.large	0.186	0.2	12	6	2	20	2	1	14.09	0.0095	8	4	14.34	0.0119	−1.75%	−20.17%
6		r5.xlarge	0.252	0.2	30	15	4	40	4	1	15.89	0.0110	16	4	15.13	0.0136	+4.99%	−19.51%
8	4	r6i.xlarge	0.252	0.2	56	28	4	40	4	1	18.18	0.0125	28	7	15.16	0.0168	+19.92%	−25.45%
12		z1d.xlarge	0.372	0.2	132	66	4	40	4	1	24.56	0.0178	20	5	17.37	0.0197	+41.37%	−10.03%
16		r5.2xlarge	0.504	0.2	240	120	8	80	8	1	27.28	0.0207	24	3	22.04	0.0229	+23.80%	−9.53%
24	8	r6i.2xlarge	0.504	0.2	552	276	8	80	8	1	41.34	0.0314	48	6	36.42	0.0532	+13.52%	−40.91%
32		z1d.2xlarge	0.744	0.2	992	496	8	80	8	1	66.86	0.0553	56	7	59.39	0.1227	+12.58%	−54.97%
48		z1d.3xlarge	1.116	0.2	2256	1128	12	120	12	1	133.64	0.1243	96	8	126.53	0.3923	+5.61%	−68.32%
64	12	z1d.3xlarge	1.116	0.2	4032	2016	12	120	12	1	236.44	0.2199	60	5	226.84	0.4922	+4.23%	−55.33%
96		z1d.3xlarge	1.116	0.2	9120	4560	12	120	12	1	496.71	0.4619	72	6	559.00	1.3863	−11.14%	−66.68%
2		r5.4xlarge	1.008	0.2	2	1	16	160	16	1	11.06	0.0100	112	7	12.00	0.0310	−7.80%	−67.84%
3	16	r6i.4xlarge	1.008	0.2	6	3	16	160	16	1	9.92	0.0089	32	2	10.45	0.0123	−5.04%	−27.58%
4		r6i.4xlarge	1.008	0.2	12	6	16	160	16	1	10.38	0.0093	144	9	12.12	0.0381	−14.39%	−75.46%
6		z1d.6xlarge	2.232	0.2	30	15	24	240	24	1	11.64	0.0144	144	6	11.83	0.0514	−1.68%	−71.91%
8	24	z1d.6xlarge	2.232	0.2	56	28	24	240	24	1	12.99	0.0161	192	8	13.79	0.0770	−5.83%	−79.07%
12		z1d.6xlarge	2.232	0.2	132	66	24	240	24	1	15.37	0.0191	216	9	16.12	0.0999	−4.63%	−80.93%
16		r5.8xlarge	2.016	0.2	240	120	32	320	32	1	21.75	0.0257	288	9	21.40	0.1211	+1.63%	−78.81%
24	32	r6i.8xlarge	2.016	0.2	552	276	32	320	32	1	35.61	0.0420	224	7	36.01	0.1635	−1.10%	−74.29%
32		r6i.8xlarge	2.016	0.2	992	496	32	320	32	1	58.45	0.0690	256	8	58.51	0.2984	−0.10%	−76.89%
48		r5.12xlarge	3.024	0.2	2256	1128	48	480	48	1	127.85	0.1867	336	7	126.89	0.8248	+0.75%	−77.37%
64	48	r6i.12xlarge	3.024	0.2	4032	2016	48	480	48	1	238.60	0.3484	48	1	236.98	0.3460	+0.68%	+0.68%
96		z1d.12xlarge	4.464	0.2	9120	4560	48	480	48	1	554.01	1.0305	432	9	559.64	6.5925	−1.01%	−84.37%

I. **Regarding the *Diff Sequences Spark* application**

* Data Input: $n = 256$ SARS-CoV-2 nucleotide sequences.

II. **Regarding the *Diff Sequences Spark*'s optimization problems**

* *Master* Nodes: $\iota_m = 1$ and $\upsilon_m = 2.232$ USD per hour of use;
* *Worker* Nodes: $\gamma_w = 4$ and $\upsilon_w = 0.372$ USD per hour of use (if on-demand instance) or $\upsilon_w = 0.1116$ USD per hour of use (if spot instance);
* Optimization Constraints: $E_{c_{min}} = \gamma_w$, $E_{c_{max}} = 8 \cdot E_{c_{min}}$, and $\phi = 6.0$ USD.

Suppose the inexperienced cloud user is willing to take the advice of the optimizer to compare $n = 256$ SARS-CoV-2 sequences with a maximum monetary spend $\phi = 6.0$ USD. Moreover, let T be the runtime cost in seconds and C be the monetary cost in USD to execute the *Diff Sequences Spark* application.

7.2.2 Costs of Rental Spot Instances Compared to On-demand Instances

The Spark cluster is initially to be formed exclusively by on-demand instances. The single *Master* node is of *z1d.6xlarge* instance type and costs $\upsilon_m = 2.232$ USD per hour of use. The *Workers* nodes are of *z1d.xlarge* instance type, and each costs $\upsilon_w = 0.372$ USD per hour of use. **Table 7** summarizes the results obtained. In this initial setup, the optimizer recommended the rental of eight on-demand *Workers* ($E_{c_{opt}} = 32$, $\iota_w = 8$), and the average runtime and monetary costs obtained after the execution were, respectively, 3411.54 seconds and 4.9354 USD.

Table 7: *Diff Sequences Spark*'s on-demand workers optimization results.

n	OPTIMIZATION INPUTS								OPTIMAL SELECTION			
									Parameters		Results	
	γ_w	ω_w	υ_w	ϕ	M	R	$E_{c_{min}}$	$E_{c_{max}}$	$E_{c_{opt}}$	ι_w	T (s)	C (USD)
256	4	z1d.xlarge	0.372	6.0	65 280	32 640	4	32	32	8	3411.54	4.9354

Next, all the *Workers* nodes are replaced by spot instances. The *Workers* are also of *z1d.xlarge* instance type, and each costs $\upsilon_w = 0.1116$ USD per hour of use. **Table 8** summarizes the results obtained. In this more economical setup, the optimizer still recommended the rental of eight *Workers* ($E_{c_{opt}} = 32$, $\iota_w = 8$), and the average runtime and monetary costs obtained after the execution were, respectively, 3411.93 seconds and 2.9616 USD.

Table 8: *Diff Sequences Spark*'s spot workers no revocation optimization results.

n	OPTIMIZATION INPUTS								OPTIMAL SELECTION			
									Parameters		Results	
	γ_w	ω_w	v_w	ϕ	M	R	$E_{c_{min}}$	$E_{c_{max}}$	$E_{c_{opt}}$	ι_w	T (s)	C (USD)
256	4	z1d.xlarge	0.1116	6.0	65 280	32 640	4	32	32	8	3411.93	2.9616

Note that while the average runtime kept equivalent, a 39.99% reduction of the monetary cost was achieved, demonstrating the benefits of using the spot instances.

7.2.3 Spot Instances Revocation Scenarios

Finally, we are interested in the primary analysis of spot instances revocation scenarios when running the *Diff Sequences Spark* application in a Spark cluster formed on AWS EC2. Consider the following revocation scenarios:

- *None*: no spot *Workers* revocation occurs. It is the baseline scenario (*i.e.*, starting point used for comparisons), which considers the average results obtained in the previous subsection for spot instances;
- RS_1: two spot *Workers* (25%) are revoked after 900 seconds of execution;
- RS_2: two spot *Workers* (25%) are revoked after 1800 seconds of execution; and
- RS_3: two spot *Workers* (25%) are revoked after 2700 seconds of execution.

Table 9 summarizes the average (Avg) and standard deviation ($StDev$) values for the runtime (T) and monetary cost (C), and the percentage change between the revocation scenarios. Notice the RS_1 scenario obtained the worst execution time average results (an increase of 0.70%) due to *Workers* loss at the beginning. On the other hand, it allowed the lowest monetary cost to run the application (a reduction of 4.61%).

Table 9: *Diff Sequences Spark*'s spot workers revocation scenarios results.

Revocation Scenario	T (s)		C (USD)		Percentage Change	
	Avg	$StDev$	Avg	$StDev$	T	C
None	3411.93	22.13	2.9616	0.0192	0%	0%
RS_1	3435.73	22.38	2.8250	0.0180	+0.70%	−4.61%
RS_2	3418.28	18.38	2.8667	0.0148	+0.19%	−3.20%
RS_3	3415.61	19.13	2.9204	0.0154	+0.11%	−1.39%

8 Concluding Remarks and Future Direction

8.1 Conclusion

This work presented a computational cost optimization model proposal for executing MapReduce-like Spark applications in the Cloud. This model aids the user in the most appropriate Spark cluster sizing, given an input parameter set that characterizes the target application (*e.g.*, number of map tasks, number of reduce tasks, and number of Executors cores) and the optimization objective (*i.e.*, time minimization subject to budget constraint or cost minimization subject to deadline constraint).

The *Diff Sequences Spark* application was developed to evaluate the proposed model. It is a bioinformatic-related application that highlights the mismatching nucleotide characters of the under comparison biological sequences. Its first version (v0.8.3[1]) aimed at DataFrames usage through a higher-level abstraction with relational programming, a more direct implementation than RDDs (NUNES et al., 2021). Two comparison approaches were implemented, $DIFF_1$ and $DIFF_{opt}$. The former allows two biological sequences to be compared per iteration (application's flow cycle), and the latter allows two or more at a time, capped by the max_S constant defined by the user. Preliminary results using multiple Worker nodes showed that $DIFF_{opt}$ achieved approximately 92.3% faster execution time when compared to $DIFF_1$, considering $n = 64$ SARS-CoV-2 input sequences and $max_S = 63$ (NUNES et al., 2021). Its second version (v0.9.9[2]) aimed at RDDs usage through a lower-level abstraction with functional programming that offered even faster sequences comparisons. Preliminary results using a single Worker node and the $DIFF_1$ approach showed that RDDs achieved approximately 30.4% execution time reduction when compared to DataFrames, considering $n = 64$ SARS-CoV-2 input sequences (NUNES et al., 2022). Application-level optimizations were also applied. For instance, multiple threads (producer-consumer pattern) on the Master node spawn simultaneous

[1] https://github.com/alan-lira/diff-sequences-spark/releases/tag/v0.8.3
[2] https://github.com/alan-lira/diff-sequences-spark/releases/tag/v0.9.9

comparison jobs to the Spark cluster and avoid idle resources. Also, a damping coefficient on each comparison job's partitions optimizes the number of tasks and reduces the runtime. Preliminary results showed an average runtime reduction of 67.69% when using the application-level optimizations (NUNES et al., 2022).

Several experiments of the *Diff Sequences Spark* application were executed to learn and assess the proposed cost prediction model, expressed as a multiple linear regression function and solved as a non-negative least squares problem. A high accuracy model was established, considering the regression metrics MAE, RMSE, and R^2 and their values, respectively, 4.048, 6.095, and 0.994. When using the trained model and bearing in mind the optimizer output, the average runtime subject to a budget constraint and the monetary cost subject to a deadline constraint were reduced by up to 3.6% and 48.67%, respectively, when compared to arbitrary resource selections by an inexperienced Cloud user.

The effect of using spot instances for Workers nodes in the runtime and monetary cost of the *Diff Sequences Spark* application was also investigated, contemplating scenarios with either spot instances revocation or not. In the tested environment with no instance revocation, the spot Workers achieved a 39.99% average reduction of the monetary cost while keeping the average runtime equivalent to their on-demand counterparts. Finally, the impacts of spot instances revocations were evaluated for the *Diff Sequences Spark* application execution, considering various revocation scenarios. Even in the worst tested case, the application ran successfully with a slight increase of 0.70% in the execution time, benefiting from Spark's efficient fault-tolerant mechanism.

8.2 Publications

The following works production occurred along the development of this dissertation:

I. Directly related to the research theme:

   (NUNES et al., 2021, 2022)

II. Indirectly related to the research theme:

   (TEYLO et al., 2021b)

8.3 Open Issues and Future Works

Taking into account the *Diff Sequences Spark* application itself, the *Cloud* environment, and the *Cost Optimizer Model* proposed in this dissertation, the following issues and future directions are of particular interest:

I. **Regarding the *Diff Sequences Spark* application's current development state**:

   a. Spark built-in transformations on *Map phase*: the current *Diff Sequences Spark* application version (v0.9.12[3]) does not support the use of the *read.textFile*, *map*, and *flatMap* transformations functions. Although the use of *Producer-Consumer* threads (multi-threading pool) reduces the application runtime, the implementation of these Spark built-in transformations would allow the data parallelism decentralized from *Master*, likely obtaining a considerable boost in application performance;

   b. *Merged Write* approach: the *MW* executes the shuffling of partial data results among the *Worker* nodes (*i.e.*, *Executors*) before writing the comparison result to the storage volume. However, the data shuffle is a costly operation for Spark. If the number of biological sequence comparisons is high or the data per sequence file is large, a significant Spark environment overhead may incur along the application execution. The *Distributed Write* (*DW*) approach combined with a *Storage as a Service* (*STaaS*) solution, *e.g.*, *Amazon S3*, might be enough to avoid excessive shuffling. For each biological sequence comparison, using *DW*, merging the partial results into one file in the remote storage would require an additional computational step. However, it should be significantly cheaper than Spark's internal procedure for large data amounts; and

   c. $DIFF_{opt}$ approach on *RDD*s: in previous work (NUNES et al., 2021), we confirmed the efficiency of the $DIFF_{opt}$ approach compared to $DIFF_1$ when running a reasonable amount of biological sequence comparisons (*i.e.*, $n = 540$ SARS-CoV-2 sequences). However, at the time, we only had *DataFrame*s implemented for the *Diff Sequences Spark* application. In the current version (v0.9.12[3]), it is already possible to use *RDD*s with the $DIFF_{opt}$ approach to run the comparisons. So it should be interesting to evaluate the efficiency of *RDD*s against *DataFrame*s for biological sequence comparisons, now both using $DIFF_{opt}$.

[3] https://github.com/alan-lira/diff-sequences-spark/releases/tag/v0.9.12

II. **Regarding the *Cloud* environment**:

a. Spot instances revocation: a time constraint might be part of an *SLA*, such that if several spot instances revocation occur, the agreement might get violated. Reactive schemes, such as the one developed by Teylo *et al.* (TEYLO et al., 2021a), should provide proper treatment in these situations.

b. Backup instances in case of Masters' failure: task assignment and management in Spark takes place exclusively in Master nodes, so there must be at least one active Master for a given application. Although replicated Master nodes increase the monetary cost, it provides greater assurance to the application execution.

c. Backup instances in case of Workers' failure: task processing in Spark takes place exclusively in Workers nodes, so there must be at least one active Worker for a given application. For example, if a spot instance type is revoked, all Workers of that type most likely will be lost. To avoid the lack of Workers problem, one could define at least one Worker instance in the on-demand market that can run the application while new spot Workers are initializing or renting different spot instance types (with unlike revocation rates) for the Worker nodes.

III. **Regarding the *MapReduce-like Spark application's cost optimizer model***:

a. Heterogeneous VMs configurations support: the optimization model proposed in this dissertation assumes that the *Master* nodes instances are homogeneous (*i.e.*, same instance type) and the *Worker* nodes instances are homogeneous. However, sometimes it is required to build a heterogeneous *Spark Cluster*, *e.g.*, due to a VM instance's lack of availability. An enhanced model would include a greater variety of configurations, with different sizes of *Spark Clusters*, satisfying new provisioning constraints; and

b. Adaptability degree evaluation: a usage evaluation of the model proposed in this dissertation to other Spark applications will allow verifying its degree of adaptability and, if necessary, improve it to obtain a more general model. Another form of evaluation concerns the degree of adaptability to other MapReduce frameworks. It is currently under investigation, initially considering the classic *Word Count* application and the *MARLA* framework (CAMPBELL et al., 2022).

REFERENCES

ABRAMOWITZ, Milton; STEGUN, Irene A. **Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables**. Washington, D.C., USA: U.S. Government Printing Office, 1964. p. 14.

AMAZON. **Amazon AWS - Global Infrastructure**. 2022. Available from: <https://aws.amazon.com/en/about-aws/global-infrastructure/>. Visited on: 30 June 2022.

_____. **Amazon AWS - Services**. 2022. Available from: <https://aws.amazon.com/en/products/>. Visited on: 17 Apr. 2022.

_____. **Amazon EC2**. 2022. Available from: <https://aws.amazon.com/ec2/>. Visited on: 17 Apr. 2022.

_____. **Amazon EC2 Instance Types**. 2022. Available from: <https://aws.amazon.com/ec2/instance-types/>. Visited on: 18 Apr. 2022.

_____. **Amazon EMR Features**. 2022. Available from: <https://aws.amazon.com/emr/features/>. Visited on: 22 July 2022.

_____. **Amazon EMR Pricing**. 2022. Available from: <https://aws.amazon.com/emr/pricing/>. Visited on: 22 July 2022.

APACHE. **Apache Hadoop**. 2022. Available from: <https://hadoop.apache.org/>. Visited on: 20 Apr. 2022.

_____. **Apache Spark**. 2022. Available from: <https://spark.apache.org/>. Visited on: 20 Apr. 2022.

_____. **Apache Spark: Concurrently Jobs Within Application**. 2022. Available from: <https://spark.apache.org/docs/latest/job-scheduling.html#overview>. Visited on: 30 June 2022.

_____. **Apache Spark: Knowing The Size of Each Task**. 2022. Available from: <https://spark.apache.org/docs/0.9.2/tuning.html#broadcasting-large-variables>. Visited on: 10 June 2022.

APACHE. **Apache Spark: Level of Parallelism**. 2022. Available from: <https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism>. Visited on: 30 June 2022.

_____. **Apache Spark: Recommended Task Size**. 2022. Available from: <https://issues.apache.org/jira/browse/SPARK-2185>. Visited on: 1 June 2022.

ARMBRUST, Michael et al. Scaling spark in the real world: performance and usability. **VLDB Endowment**, VLDB Endowment, v. 8, n. 12, p. 1840–1843, 2015. DOI: 10.14778/2824032.2824080.

ARMBRUST, Michael et al. Spark SQL: Relational Data Processing in Spark. In: PROCEEDINGS of the 2015 ACM SIGMOD International Conference on Management of Data. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015. (SIGMOD '15), p. 1383–1394. DOI: 10.1145/2723372.2742797.

ARORA, Sanjeev; BARAK, Boaz. **Computational Complexity: A Modern Approach**. Cambridge, United Kingdom: Cambridge University Press, 2009. p. 27–43. DOI: 10.1017/CBO9780511804090.

BAHRAMI, Afsane; FERNS, Gordon A. Genetic and pathogenic characterization of SARS-CoV-2: a review. **Future Virology**, Future Medicine, v. 15, n. 8, p. 533–549, 2020. DOI: 10.2217/fvl-2020-0129.

BERNSTEIN, David. Containers and Cloud: From LXC to Docker to Kubernetes. **IEEE Cloud Computing**, v. 1, n. 3, p. 81–84, 2014. DOI: 10.1109/MCC.2014.51.

BOEHM, Matthias et al. SystemML: declarative machine learning on spark. **VLDB Endowment**, VLDB Endowment, v. 9, n. 13, p. 1425–1436, 2016. DOI: 10.14778/3007263.3007279.

BONCZ, Peter et al. MonetDB/XQuery—Consistent and Efficient Updates on the Pre/Post Plane. **International Conference on Extending Database Technology**, v. 3896, p. 1190–1193, 2006. DOI: 10.1007/11687238_89.

BOTCHKAREV, Alexei. A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms. **Interdisciplinary Journal of Information, Knowledge, and Management**, Informing Science Institute, v. 14, p. 45–76, 2019. DOI: 10.28945/4184.

BRUM, Rafaela et al. A Fault Tolerant and Deadline Constrained Sequence Alignment Application on Cloud-Based Spot GPU Instances. **European Conference on Parallel Processing**, Springer International Publishing, p. 317–333, 2021. DOI: 10.1007/978-3-030-85665-6_20.

CAMPBELL, Ronald et al. MapReduce na AWS: Uma Análise de Custos Computacionais Utilizando os Serviços FaaS e IaaS. **XXIII Symposium in High Performance Computing Systems**, SBC, Florianópolis, SC, Brazil, 2022. (Under Review).

CHANNELE2E. **Cloud Market Share 2022: Amazon AWS, Microsoft Azure, Google Cloud.** 2022. Available from: <https://www.channele2e.com/news/cloud-market-share-amazon-aws-microsoft-azure-google/>. Visited on: 10 June 2022.

CHEN, Donghui; PLEMMONS, Robert J. Nonnegativity Constraints in Numerical Analysis. In: THE Birth of Numerical Analysis. Hackensack, New Jersey, USA: World Scientific, 2010. p. 109–139. DOI: 10.1142/9789812836267_0008.

CHEN, Keke et al. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 25, n. 6, p. 1403–1412, June 2014. DOI: 10.1109/TPDS.2013.297.

CSC, Coronaviridae Study Group. The species Severe acute respiratory syndrome-related coronavirus: classifying 2019-nCoV and naming it SARS-CoV-2. **Nature Microbiology**, Nature Publishing Group UK London, v. 5, n. 4, p. 536–544, 2020. DOI: 10.1038/s41564-020-0695-z.

DANTZIG, George B. **Origins of the Simplex Method**. New York, NY, USA: Association for Computing Machinery, 1990. p. 141–151. DOI: 10.1145/87252.88081.

DE OLIVEIRA, Douglas et al. Towards optimizing the execution of spark scientific workflows using machine learning-based parameter tuning. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 33, n. 5, 2021. DOI: 10.1002/cpe.5972.

DIJKSTRA, Edsger W. A first exploration of effective reasoning. Department of Computer Sciences. University of Texas at Austin. Austin, TX, USA, July 1996. Available from: <http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1239.PDF>. Visited on: 30 June 2022.

DOMO. **Data never sleeps.** 2020. Available from: <https://www.domo.com/solution/data-never-sleeps-6>. Visited on: 1 Feb. 2022.

DROSTEN, Christian et al. Identification of a Novel Coronavirus in Patients with Severe Acute Respiratory Syndrome. **New England Journal of Medicine (NEJM)**, Massachusetts Medical Society, v. 348, n. 20, p. 1967–1976, 2003. DOI: 10.1056/nejmoa030747.

DURBIN, R. et al. **Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids**. Cambridge, UK: Cambridge University Press, 1998.

ENCYCLOPÆDIA BRITANNICA. **Biomolecule Definition**. 2022. Available from: <https://www.britannica.com/science/biomolecule>. Visited on: 4 July 2022.

_____. **DNA Sequencing**. 2022. Available from: <https://www.britannica.com/science/DNA-sequencing>. Visited on: 4 July 2022.

_____. **Sequencing and bioinformatic analysis of genomes**. 2022. Available from: <https://www.britannica.com/science/genomics#ref974394>. Visited on: 4 July 2022.

FOSTER, Ian; GANNON, Dennis B. **Cloud Computing for Science and Engineering**. Cambridge, MA, USA: MIT Press, 2017.

GISAID. **GISAID - Global Initiative on Sharing Avian Influenza Data.** 2022. Available from: <https://gisaid.org/>. Visited on: 20 Feb. 2022.

GRAHAM, Ronald L.; KNUTH, Donald E.; PATASHNIK, Oren. Binomial Coefficients. In: CONCRETE Mathematics: A Foundation for Computer Science. Boston, Massachusetts, USA: Addison-Wesley, 1994. p. 153–154.

GRANDL, Robert et al. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. **12th USENIX Conference on Operating Systems Design and Implementation**, USENIX Association, Savannah, GA, USA, p. 81–97, 2016.

GUROBI. **Gurobi Optimizer**. 2022. Available from: <https://www.gurobi.com/products/gurobi-optimizer/>. Visited on: 3 May 2022.

HAIR, Joseph F. et al. **Multivariate Data Analysis**. Andover, Hampshire, United Kingdom: Cengage Learning, 2018.

HATCHER, Eneida L. et al. Virus Variation Resource – improved response to emergent viral outbreaks. **Nucleic Acids Research**, Oxford Academy, v. 45, n. D1, p. d482–d490, Jan. 2017. DOI: 10.1093/nar/gkw1065.

HELLEWELL, Joel et al. Feasibility of controlling COVID-19 outbreaks by isolation of cases and contacts. **The Lancet Global Health**, Elsevier, v. 8, n. 4, e488–e496, 2020. DOI: 10.1016/S2214-109X(20)30074-7.

HEY, Anthony JG; TANSLEY, Stewart; TOLLE, Kristin Michele. **The Fourth Paradigm: Data-intensive Scientific Discovery**. Redmond, WA, USA: Microsoft Research Lab, 2009. ISBN 978-0-9825442-0-4. Available from: <http://research.microsoft.com/en-us/collaboration/fourthparadigm/>.

HINDMAN, Benjamin et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In: PROCEEDINGS of the 8th USENIX Conference on Networked Systems Design and Implementation. Boston, MA: USENIX Association, 2011. (NSDI'11), p. 295–308.

HU, Han et al. Toward Scalable Systems for Big Data Analytics: A Technology Tutorial. **IEEE Access**, IEEE, v. 2, p. 652–687, 2014. DOI: 10.1109/ACCESS.2014.2332453.

ISLAM, Muhammed Tawfiqul et al. Cost-efficient Dynamic Scheduling of Big Data Applications in Apache Spark on Cloud. **Journal of Systems and Software**, Elsevier, v. 162, 2020. DOI: 10.1016/j.jss.2019.110515.

ISLAM, Muhammed Tawfiqul et al. SLA-based Scheduling of Spark Jobs in Hybrid Cloud Computing Environments. **IEEE Transactions on Computers**, IEEE, v. 71, n. 5, p. 1117–1132, 2021. DOI: 10.1109/TC.2021.3075625.

JYOTHI, Sangeetha Abdu et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. **12th USENIX Conference on Operating Systems Design and Implementation**, USENIX Association, Savannah, GA, USA, p. 117–134, 2016.

KARANASOS, Konstantinos; SURESH, Arun; DOUGLAS, Chris. Advancements in YARN Resource Manager. In: ENCYCLOPEDIA of Big Data Technologies. Washington, DC, USA: Springer International Publishing, 2018. p. 1–9. DOI: 10.1007/978-3-319-63962-8_207-1.

KRANAS, Pavlos et al. Parallel query processing in a polystore. **Distributed and Parallel Databases**, Springer, v. 39, n. 4, p. 939–977, 2021. DOI: 10.1007/s10619-021-07322-5.

KULKARNI, Apurva; RAMANATHAN, Chandrashekar. HS-PARAM: Hive-Spark Parameterization Framework to Optimize Ingestion and Storage of Heterogeneous Data. **14th International Conference on Communication Systems & Networks**, Bangalore, India, p. 227–230, 2022. DOI: 10.1109/COMSNETS53615.2022.9668594.

LAU, Billy T. et al. Profiling SARS-CoV-2 mutation fingerprints that range from the viral pangenome to individual infection quasispecies. **Genome Medicine**, BioMed Central, v. 13, n. 1, p. 1–23, 2021. DOI: 10.1186/s13073-021-00882-2.

LAWSON, Charles L.; HANSON, R.J. Linear Least Squares with Linear Inequality Constraints. In: SOLVING Least Squares Problems. Philadelphia, USA: Society for Industrial & Applied Mathematics, 1987. chap. 23, p. 158–173. DOI: 10.1137/1.9781611971217.ch23.

LI, Min et al. SparkBench: A Spark Benchmarking Suite Characterizing Large-Scale in-Memory Data Analytics. **Cluster Computing**, Kluwer Academic Publishers, USA, v. 20, n. 3, p. 2575–2589, 2017. DOI: 10.1007/s10586-016-0723-1.

MAKRIS, Antonios et al. MongoDB Vs PostgreSQL: A comparative study on performance aspects. **GeoInformatica**, Springer, v. 25, n. 2, p. 243–268, 2021. DOI: 10.1007/s10707-020-00407-w.

MATHWORKS. **MATLAB lsqnonneg**. 2022. Available from: <https://www.mathworks.com/help/matlab/ref/lsqnonneg.html>. Visited on: 10 May 2022.

MELL, Peter; GRANCE, Timothy. The NIST Definition of Cloud Computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards & Technology (NIST), Gaithersburg, Maryland, USA, 2011. DOI: 10.6028/NIST.SP.800-145.

MENG, Xiangrui et al. MLlib: Machine Learning in Apache Spark. **The Journal of Machine Learning Research**, JMLR.org, v. 17, n. 1, p. 1235–1241, Jan. 2016.

NAQVIA, Ahmad Abu Turab et al. Insights into SARS-CoV-2 genome, structure, evolution, pathogenesis and therapies: Structural genomics approach. **Biochimica et Biophysica Acta (BBA) - Molecular Basis of Disease**, Elsevier, v. 1866, n. 10, 2020. DOI: 10.1016/j.bbadis.2020.165878.

NCBI. **NCBI - National Center for Biotechnology Information.** 2022. Available from: <https://www.ncbi.nlm.nih.gov/>. Visited on: 20 Feb. 2022.

_____. **NCBI SARS-CoV-2 Resources**. 2022. Available from: <https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/virus?SeqType_s=Nucleotide&VirusLineage_ss=Severe%20acute%20respiratory%20syndrome%20coronavirus%202,%20taxid:2697049&HostLineage_ss=Homo%20sapiens%20(human),%20taxid:9606>. Visited on: 5 July 2022.

NHGRI. **Deoxyribonucleic Acid (DNA) Definition**. 2022. Available from: <https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid>. Visited on: 4 July 2022.

NHGRI. **Nucleotide Definition**. 2022. Available from: <https://www.genome.gov/genetics-glossary/Nucleotide>. Visited on: 4 July 2022.

_____. **Ribonucleic Acid (RNA) Definition**. 2022. Available from: <https://www.genome.gov/genetics-glossary/RNA-Ribonucleic-Acid>. Visited on: 4 July 2022.

NUNES, A.L. et al. Optimizing Computational Costs of Spark for SARS-CoV-2 Sequences Comparisons on a Commercial Cloud. **Concurrency and Computation: Practice and Experience**, 2022. (Under Review).

NUNES, A.L. et al. Towards Analyzing Computational Costs of Spark for SARS-CoV-2 Sequences Comparisons on a Commercial Cloud. **XXII Symposium in High Performance Computing Systems**, SBC, Belo Horizonte, MG, Brazil, p. 192–203, 2021. DOI: 10.5753/wscad.2021.18523.

PAL, Surajit; GAURI, Susanta Kumar. Assessing effectiveness of the various performance metrics for multi-response optimization using multiple regression. **Computers & Industrial Engineering**, v. 59, n. 4, p. 976–985, 2010. DOI: 10.1016/j.cie.2010.09.009.

PAPADIMITRIOU, Christos H.; STEIGLITZ, Kenneth. **Combinatorial Optimization: Algorithms and Complexity**. Mineola, New York, USA: Dover Publications, 1998.

PERERA, Shelan; PERERA, Ashansa; HAKIMZADEH, Kamal. Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds. **arXiv**, 2016. DOI: 10.48550/arXiv.1610.04493.

PEREZ, Tiago BG; ZHOU, Xiaobo; CHENG, Dazhao. Reference-distance Eviction and Prefetching for Cache Management in Spark. **47th International Conference on Parallel Processing**, ACM, Eugene, OR, USA, p. 1–10, 2018. DOI: 10.1145/3225058.3225087.

ROTHLAUF, Franz. **Optimization Methods**. Berlin, Germany: Springer International Publishing, 2011. p. 45–102. DOI: 10.1007/978-3-540-72962-4_3.

SCIKIT-LEARN. **sklearn.metrics**. 2022. Available from: <https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics>. Visited on: 13 May 2022.

SCIPY. **optimize nnls**. 2022. Available from: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.nnls.html>. Visited on: 10 May 2022.

TEYLO, Luan et al. A dynamic task scheduler tolerant to multiple hibernations in cloud environments. **Cluster Computing**, Springer International Publishing, v. 24, n. 2, p. 1051–1073, 2021. DOI: 10.1007/s10586-020-03175-2.

TEYLO, Luan et al. Comparing SARS-CoV-2 Sequences using a Commercial Cloud with a Spot Instance Based Dynamic Scheduler. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). Melbourne, Australia: IEEE, 2021. p. 247–256. DOI: 10.1109/CCGrid51090.2021.00034.

VAN DE GEER, Sara Anna. Least Squares Estimation. In: ENCYCLOPEDIA of Statistics in Behavioral Science. Chichester, West Sussex, England: John Wiley & Sons, 2005. v. 2, p. 1041–1045. DOI: 10.1002/0470013192.

VAQUERO, Luis Miguel; RODERO-MERINO, Luis; BUYYA, Rajkumar. Cloud scalability: building the Millennium Falcon. **Concurrency and Computation: Practice and Experience**, v. 25, n. 12, p. 1623–1627, 2013. DOI: 10.1002/cpe.3008.

WANG, Lei et al. BigDataBench: A big data benchmark suite from internet services. **IEEE 20th International Symposium on High Performance Computer Architecture**, IEEE, p. 488–499, 2014. DOI: 10.1109/HPCA.2014.6835958.

WEISS, Susan R.; NAVAS-MARTIN, Sonia. Coronavirus Pathogenesis and the Emerging Pathogen Severe Acute Respiratory Syndrome Coronavirus. **Microbiology and Molecular Biology Reviews**, American Society for Microbiology Journals, v. 69, n. 4, p. 635–664, 2005. DOI: 10.1128/MMBR.69.4.635-664.2005.

WU, Fan et al. A new coronavirus associated with human respiratory disease in China. **Nature**, Nature Publishing Group, v. 579, n. 7798, p. 265–269, 2020. DOI: 10.1038/s41586-020-2008-3.

XIN, Reynold S et al. GraphX: A Resilient Distributed Graph System on Spark. **First International Workshop on Graph Data Management Experiences and Systems**, ACM, New York, NY, USA, p. 1–6, 2013. DOI: 10.1145/2484425.2484427.

XU, Bo et al. DSA: Scalable Distributed Sequence Alignment System Using SIMD Instructions. **17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**, IEEE, p. 758–761, 2017. DOI: 10.1109/CCGRID.2017.74.

XU, Bo et al. Efficient Distributed Smith-Waterman Algorithm Based on Apache Spark. **IEEE 10th International Conference on Cloud Computing**, IEEE, p. 608–615, 2017. DOI: 10.1109/CLOUD.2017.83.

XU, Fei et al. Cost-Effective Cloud Server Provisioning for Predictable Performance of Big Data Analytics. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 30, n. 5, p. 1036–1051, 2018. DOI: 10.1109/TPDS.2018.2873397.

XU, Yinggen; LIU, Liu; DING, Zhijun. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. **IEEE International Parallel and Distributed Processing Symposium**, IEEE, p. 378–387, 2020. DOI: 10.1109/IPDPS47924.2020.00047.

YAN, Ying et al. TR-Spark: Transient Computing for Big Data Analytics. **17th ACM Symposium on Cloud Computing**, ACM, Santa Clara, CA, USA, p. 484–496, 2016. DOI: 10.1145/2987550.2987576.

YOUSEFF, Lamia; BUTRICO, Maria; DA SILVA, Dilma. Toward a Unified Ontology of Cloud Computing. In: GRID Computing Environments Workshop (GCE). Austin, Texas, USA: IEEE, 2008. p. 1–10. DOI: 10.1109/GCE.2008.4738443.

ZAHARIA, Matei et al. Apache Spark: A Unified Engine for Big Data Processing. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 11, p. 56–65, 2016. DOI: 10.1145/2934664.

ZAHARIA, Matei et al. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. **4th USENIX Conference on Hot Topics in Cloud Computing**, USENIX Association, Boston, MA, USA, 2012.

ZAHARIA, Matei et al. 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. San Jose, CA, USA: USENIX Association, 2012. (NSDI '12), p. 15–28.

ZAKI, Ali M. et al. Isolation of a Novel Coronavirus from a Man with Pneumonia in Saudi Arabia. **New England Journal of Medicine**, Massachusetts Medical Society, v. 367, n. 19, p. 1814–1820, 2012. DOI: 10.1056/NEJMoa1211721.

ZHAO, Guoguang; LING, Cheng; SUN, Donghong. SparkSW: Scalable Distributed Computing System for Large-Scale Biological Sequence Alignment. **15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**, IEEE, p. 845–852, 2015. DOI: 10.1109/CCGrid.2015.55.

APPENDIX A — D_a's Estimation Function Evaluation

In Dijkstra's own words (DIJKSTRA, 1996), "*A picture may be worth a thousand words, a formula is worth a thousand pictures.*"

Subsection 3.2.1 (*Diff Sequences Spark*'s implementation details section) presented a mathematical formula that outputs the estimation value of the number of biological sequence comparisons that will be processed, denoted as D_a. Given the number of unique sequences (n), where each sequence seq_i must be compared against all the others without repetitions, and given the maximum simultaneous sequences that are compared against seq_i at a time (max_S), D_a can be estimated as follows:

$$D_a = \begin{cases} \left\lceil \frac{n}{max_S} \cdot \left((n-1) - \left(\frac{n - max_S}{2}\right) \right) \right\rceil + \epsilon, & \text{if } 1 \leq max_S < \frac{n}{2} \\ 2 \cdot \left((n-1) - \left(\frac{max_S}{2}\right) \right), & \text{if } \frac{n}{2} \leq max_S < n \end{cases} \tag{A.1}$$

$$\text{where } n, max_S \in \mathbb{Z} \text{ and } n \geq 2$$

This piecewise-defined function (**Function A.1**) was built from the empirical analysis of a small set of n and max_S values combinations, where $n \in [2, 10]$ and $max_S \in [1, n[$, having 45 elements in total. As presented by Abramowitz & Stegun (ABRAMOWITZ; STEGUN, 1964), if x_0 is an approximation to the true value of x, then:

(a) The *Absolute Error* of x_0 is $\Delta x = x_0 - x$. So, $x - x_0$ is the correction to x. It is the difference between the approximation value x_0 and its true value x;

(b) The *Relative Error* of x_0 is $\delta x = \frac{\Delta x}{x} = \frac{x_0 - x}{x} = \frac{x_0}{x} - 1$. It is the ratio of Δx to the true value x, which indicates how good the approximation value x_0 is relative to the size of the true value x; and

(c) The *Percentage Error* is 100 times the relative error. It expresses δx as a value between 0 and 100 rather than as a fraction.

The estimation values of D_a were confronted with their actual values through the above error analysis metrics, mainly concerning the magnitude (modulus) of the relative error. An approximation error ϵ (Δx) was observed in 6 of the 20 combinations that fitted into the first equation, *i.e.*, when $max_S \in [1, \frac{n}{2}[$. A minimum relative error of 0.05 (5.0%) was obtained with the $\{n = 9, max_S = 2\}$ combination, which represents the best-case approximation error. A maximum relative error of 0.1667 (16.67%) was obtained with the $\{n = 5, max_S = 2\}$ combination, which represents the worst-case approximation error. The average relative error was 0.0917 (9.17%) with a 0.0429 standard deviation for these six combinations. Furthermore, none of the 25 combinations that fitted into the second equation experienced approximation errors, *i.e.*, when $max_S \in [\frac{n}{2}, n[$.

To extrapolate the evaluation of D_a's estimated values accuracy, larger combinations sets of n and max_S were formed, as summarized in **Table 10**. Each combination set shows its respective number of combinations, the number of approximation error occurrences in both equations of **Function A.1**, and the minimum, maximum, average, and standard deviation of the relative error (δx) for the first equation.

Table 10: Evaluation of the D_a's estimation function.

Ranges		Number of Combinations	FIRST EQUATION $max_S \in [1, \frac{n}{2}[$						SECOND EQUATION $max_S \in [\frac{n}{2}, n[$	
			Occurrences		Relative Error (δx)				Occurrences	
n	max_S	$\frac{MAX(n)!}{2! \cdot (MAX(n)-2)!}$	Error-Free	With ϵ Error	Min.	Max.	Avg.	Std. Dev.	Error-Free	With ϵ Error
$[2, 10]$	$[1, n[$	45	14	6	0.05	0.1667	0.0917	0.0429	25	0
$[2, 1000]$	$[1, n[$	499 500	25 799	223 701	$4.008 \cdot 10^{-6}$	0.1667	0.0099	0.0089	250 000	0
$[2, 2000]$	$[1, n[$	1 999 000	60 016	938 984	$1.001 \cdot 10^{-6}$	0.1667	0.0098	0.0091	1 000 000	0
$[2, 3000]$	$[1, n[$	4 498 500	97 407	2 151 093	$4.447 \cdot 10^{-7}$	0.1667	0.0098	0.0091	2 250 000	0
$[2, 4000]$	$[1, n[$	7 998 000	136 523	3 861 477	$2.501 \cdot 10^{-7}$	0.1667	0.0098	0.0092	4 000 000	0
$[2, 5000]$	$[1, n[$	12 497 500	177 289	6 070 211	$1.601 \cdot 10^{-7}$	0.1667	0.0097	0.0092	6 250 000	0

Notice the first row contains the results for the combinations set used in the D_a's function building process, and the subsequent rows show the extrapolated results. Although the occurrences of ϵ errors increase for larger combinations set for the first equation, their relative error's minimum, average, and standard deviation are significantly reduced. So, D_a's estimation function is still helpful for higher ranges and achieves low average relative errors. Once again, no approximation errors ϵ occurred for the second equation.

Lastly, the implementation for D_a's estimation and analysis of its respective approximation errors is available *here*[1], and the evaluation results data are available *here*[2].

[1] https://github.com/alan-lira/biological-sequences-comparisons-number-estimator
[2] https://doi.org/10.17605/OSF.IO/TDEPK

APPENDIX B — *Diff Sequences Spark*'s Application-Level Optimizations Evaluation

Subsection 3.2.4 (*Diff Sequences Spark*'s implementation details section) presented some runtime improvement proposals. Consider the following optimization scenarios:

- OS_1: the *Master* node submits one sequences comparison job per time to the Spark cluster through the default *FIFO* scheduling mode (unique active job within the application); and
- OS_2: the *Master* node, composed of 12 producer and 12 consumer threads, submits multiple sequences comparison jobs to the Spark cluster through the *FAIR* Standalone scheduler, which employs a single fair resource sharing pool.

Moreover, consider the following general settings:

I. **Regarding the experimental environment (Spark on AWS EC2)**

   * AWS EC2 Memory Optimized Instances: the *z1d.6xlarge* instance (24 CPU cores and 192 GiB RAM) for the *Master* node, and the {*z1d.large*, *z1d.xlarge*, *z1d.2xlarge*} subset of instances for the *Worker* nodes. The memory optimized offers the best cost-benefit execution (NUNES et al., 2021);
   * Spark Cluster: single *Master* node and {1, 2, 4} *Worker* nodes, with a single *Executor* per *Worker*.

II. **Regarding the *Diff Sequences Spark* application**

* Data Input: $n = 64$ SARS-CoV-2 nucleotide sequences;
* Spark's Data Structure: *RDD*;
* Diff Phase: $DIFF_1$;
* Collection Phase: *MW*;
* Optimizations: damping coefficient k_i and multi-threading pool composed of 12 producer and 12 consumer threads in the *Master*, with a maximum capacity of 300 ready-to-run comparison jobs in the shared queue.

Let ω_w be the Worker instance name, ι_w the number of Workers per experiment, E_c the total number of Executors cores (*default parallelism*), K the set of all the divisors of E_c, where $K = \{k \in \mathbb{N}^* \mid k \setminus E_c\}$, E_m the size in GiB of Executors memory, n the number of distinct biological sequences, D_a the total number of sequence comparisons to be processed, where $D_a = \frac{n \cdot (n-1)}{2}$, k_i the damping coefficient, M the total number of map tasks, where $M = 2 \cdot \frac{E_c}{k_i} \cdot D_a$, R the total number of reduce tasks, where $R = D_a$, and T the runtime in seconds of each experiment.

Table 11 summarizes the advantages of using the OS_2 optimization scenario instead of OS_1. Regarding the optimal damping coefficient k_{opt} obtained for each scenario, *i.*) for OS_1, if $E_c \leq 4$, then $k_{opt} = E_c$; otherwise, $k_{opt} = \frac{E_c}{4}$; and *ii.*) for OS_2, $k_{opt} = E_c$. The average runtime reductions when using k_{opt} instead of any $k_i \in K \setminus \{k_{opt}\}$ are 24.34% and 27.66% for OS_1 and OS_2, respectively. The average runtime reduction after switching from OS_1 to OS_2 is 67.69%.

Table 11: *Diff Sequences Spark*'s application-level optimization results.

Worker ω_w	**Node** ι_w	E_c	E_m	n	D_a	k_i	M	R	**Runtime** T **(s)** OS_1	OS_2	**Percentage Change**
z1d.large	1	2	14	64	2016	1	8064	2016	1076.72	610.89	−43.26%
						2	4032		982.75	549.57	−44.08%
	2	4	28	64	2016	1	16 128	2016	948.00	381.37	−59.77%
						2	8064		871.14	329.21	−62.21%
						4	4032		849.16	294.21	−65.35%
	4	8	56	64	2016	1	32 256	2016	1017.91	352.04	−65.41%
						2	16 128		763.86	240.47	−68.52%
						4	8064		772.54	231.01	−70.10%
						8	4032		843.56	230.11	−72.72%
z1d.xlarge	1	4	29	64	2016	1	16 128	2016	1038.74	376.92	−63.71%
						2	8064		928.05	322.38	−65.26%
						4	4032		908.57	289.17	−68.17%
	2	8	58	64	2016	1	32 256	2016	992.73	347.57	−64.99%
						2	16 128		766.04	245.75	−67.92%
						4	8064		778.99	238.12	−69.43%
						8	4032		817.38	229.93	−71.87%
	4	16	116	64	2016	1	64 512	2016	1028.86	350.30	−65.95%
						2	32 256		869.42	244.15	−71.92%
						4	16 128		696.17	229.79	−66.99%
						8	8064		711.10	223.68	−68.54%
						16	4032		795.54	215.47	−72.92%
z1d.2xlarge	1	8	61	64	2016	1	32 256	2016	1229.70	336.65	−72.62%
						2	16 128		784.19	241.51	−69.20%
						4	8064		903.31	237.40	−73.72%
						8	4032		799.22	234.09	−70.71%
	2	16	122	64	2016	1	64 512	2016	1021.47	334.92	−67.21%
						2	32 256		856.59	242.65	−71.67%
						4	16 128		697.71	237.32	−65.99%
						8	8064		710.44	233.31	−67.16%
						16	4032		782.24	214.77	−72.54%
	4	32	244	64	2016	1	129 024	2016	1131.13	259.35	−77.07%
						2	64 512		892.91	234.08	−73.78%
						4	32 256		849.94	232.15	−72.69%
						8	16 128		686.30	214.18	−68.79%
						16	8064		716.54	209.08	−70.82%
						32	4032		795.57	208.76	−73.76%