UNIVERSIDADE FEDERAL FLUMINENSE

RONALD MACHADO CAMPBELL JUNIOR

Análise de Execução de Aplicação MapReduce na AWS Utilizando os Serviços FaaS e IaaS

NITERÓI

2023

RONALD MACHADO CAMPBELL JUNIOR

Análise de Execução de Aplicação MapReduce na AWS Utilizando os Serviços FaaS e IaaS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Orientador:
LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

Coorientador:
MARIA CRISTINA SILVA BOERES

NITERÓI

2023

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

C187a Campbell Junior, Ronald Machado
Análise de Execução de Aplicação MapReduce na AWS Utilizando os Serviços FaaS e IaaS / Ronald Machado Campbell Junior. - 2023.
63 f.: il.

Orientador: Lúcia Maria de Assumpção Drummond.
Coorientador: Maria Cristina Silva Boeres.
Dissertação (mestrado)-Universidade Federal Fluminense, Instituto de Computação, Niterói, 2023.

1. Computação em nuvem. 2. Sistemas paralelos e distribuídos. 3. Produção intelectual. I. Drummond, Lúcia Maria de Assumpção, orientador. II. Boeres, Maria Cristina Silva, coorientador. III. Universidade Federal Fluminense. Instituto de Computação.IV. Título.

CDD - XXX

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

RONALD MACHADO CAMPBELL JUNIOR

ANÁLISE DE EXECUÇÃO DE APLICAÇÃO MAPREDUCE NA AWS UTILIZANDO OS SERVIÇOS FAAS E IAAS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação.

Aprovado em dezembro de 2023

BANCA EXAMINADORA

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

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

Prof. D.Sc. Alfredo Goldman vel Lejbman, USP

Prof. D.Sc. Leandro Santiago de Araújo, UFF

Prof. D.Sc. Eugene Francis Vinod Rebello, UFF

Niterói

2023

Resumo

Em busca de eficiência na execução de aplicações do tipo MapReduce, que processam grandes volumes de dados, e visando reduzir os custos monetários, a utilização de serviços de computação em nuvem se tornou uma opção bem estabelecida. Este estudo teve como objetivo apresentar os comportamentos de diferentes serviços de nuvem, através de uma análise exploratória dos tempos e custos das execuções de uma aplicação MapReduce, com foco na nuvem pública da Amazon, a AWS. Para realizar essa análise, foram implementadas aplicações MapReduce utilizando os *frameworks* Spark e MARLA, executados nos serviços EC2 e Lambda da AWS, respectivamente. Durante o estudo foi avaliado o impacto das escolhas de recursos, nos tempos de execução e custos. Os experimentos foram conduzidos em diversos cenários, variando os recursos disponíveis, utilizando a aplicação clássica de *wordcount*. Os resultados revelaram que o ambiente MARLA Lambda proporcionou a execução mais rápida, enquanto o Spark EC2 se destacou por ser mais econômico em termos de custos. Essas descobertas oferecem *insights* valiosos para a seleção de serviços em nuvem, equilibrando desempenho e economia, de acordo com as necessidades específicas de cada aplicação MapReduce.

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

Abstract

In pursuit of efficiency in the execution of MapReduce-type applications that process large volumes of data while aiming to reduce costs, the use of cloud computing services has become a well-established option. This study aimed to present the behaviors of different cloud services through an exploratory analysis of the execution times and costs of a MapReduce application, with a focus on Amazon's public cloud, AWS. To conduct this analysis, MapReduce applications were implemented using the Spark and MARLA frameworks, running on AWS services EC2 and Lambda, respectively. During the study, the impact of resource choices on execution times and costs was assessed. Experiments were conducted in various scenarios, varying the available resources, using the classic word count application. The results revealed that the MARLA Lambda environment provided the fastest execution, while Spark EC2 stood out as more cost-effective. These findings offer valuable insights for cloud service selection, balancing performance and economy, according to the specific needs of each MapReduce application.

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

Lista de Figuras

1 *Arquitetura do MARLA na AWS* 32

Lista de Tabelas

1 Recursos alocados por tipo de instância. 23

2 Trabalhos relacionados. 30

3 Quantidade mínima de mappers por memória definida. 44

4 2GB - Tempos e custos médios de execução do *Word-Count-MARLA*. . . . 47

5 4GB - Tempos e custos médios de execução do *Word-Count-MARLA*. . . . 48

6 Tempos médios de execução *Word-Count-MARLA* 2GB e 4GB - comparação. 50

7 10GB - Tempos e custos médios de execução do *Word-Count-MARLA*. . . 51

8 Tempos médios de execução *Word-Count-MARLA* 4GB e 10GB - comparação. 52

9 ArqBase - Tempos e custos médios de execução do *Word-Count-MARLA*. . 54

10 Arq25 - Tempos e custos médios de execução do *Word-Count-MARLA*. . . 54

11 Arq50 - Tempos e custos médios de execução do *Word-Count-MARLA*. . . 54

12 Arq75 - Tempos e custos médios de execução do *Word-Count-MARLA*. . . 55

13 Tempos e custos médios de execução do *Word-Count-Spark*. 56

14 MARLA Vs. Spark: melhores tempos médios de execução. 57

Sumário

1 Introdução **12**

2 Conceitos Preliminares **15**

2.1 Modelo MapReduce 15

2.1.1 Apache Hadoop 16

2.1.2 Apache Spark 17

2.2 Nuvens Computacionais 18

2.2.1 Serviços da AWS 18

2.2.2 Mercados da AWS 20

2.2.3 Famílias de Máquinas Virtuais da AWS 21

2.3 Computação sem servidor (*Serverless Computing*) 23

2.3.1 Os Benefícios da Computação sem Servidor 23

2.3.2 Desafios da computação sem servidor 24

3 Trabalhos Relacionados **27**

4 O Ambiente de análise **32**

4.1 Arquitetura do MARLA 32

4.1.1 COORDINATOR 33

4.1.2 MAPPER 35

4.1.3 REDUCER 36

4.2 MARLA - Considerações e Configurações 36

4.2.1 Arquivo de configuração 37

4.2.2 Tratamento de falhas 38

4.2.3 Considerações e Limitações 39

4.3 Implementação da Aplicação *Word Count* 39

4.3.1 MARLA - Implementação 39

4.3.2 Spark - Implementação 40

5 Resultados Experimentais na AWS **42**

5.1 Análise dos Parâmetros de Execução Configuráveis 43

5.2 Análise dos Resultados com MARLA 46

5.2.1 Execuções do *Word-Count-MARLA* com entrada de 2GB de dados . 46

5.2.2 Execuções do *Word-Count-MARLA* com entrada de 4GB de dados . 49

5.2.3 Execuções com entrada 10GB 51

5.2.4 Impacto do Conteúdo da Entrada 53

5.3 Análise dos Resultados com Spark 55

5.4 Desempenho e Custo: *Word-Count* MARLA e Spark 57

6 Conclusões e Trabalhos Futuros **59**

REFERÊNCIAS **61**

1 Introdução

A adoção crescente dos ambientes de nuvem computacional é uma realidade inegável, especialmente no que diz respeito ao compartilhamento de dados e à oferta de serviços de uso geral. Tanto provedores quanto clientes têm investido recursos consideráveis, como exibido no estudo em (MUNISWAMAIAH et al., 2019), para consolidá-la como uma solução altamente promissora na execução de aplicativos voltados para o processamento de grandes volumes de dados, também conhecidos como *Big Data*.

Este paradigma computacional oferece uma série de vantagens significativas em comparação com as infraestruturas dedicadas. Isso se reflete, sobretudo, na notável redução dos custos operacionais associados a fatores como consumo de energia, aquisição de licenças de software e obsolescência de hardware. Como resultado, os clientes podem concentrar seus esforços no desenvolvimento e implantação de aplicativos, sem a necessidade de lidar com a complexidade do gerenciamento e manutenção da infraestrutura de computação. Os provedores de serviços de nuvem disponibilizam infraestrutura, plataformas e software por meio de acordos de nível de serviço (SLAs) estabelecidos com os consumidores.

Em contrapartida, há desafios significativos a serem enfrentados quando se busca equiparar o desempenho alcançado em infraestruturas dedicadas e ambientes de nuvem. Um exemplo notável é a sobrecarga de recursos, como CPU, memória, rede e armazenamento, que surge devido à virtualização e à arquitetura *multi-tenant* (*i.e.* onde diversos clientes compartilham os recursos de um único servidor físico). Essa sobrecarga pode impactar adversamente a eficiência das aplicações de Big Data quando executadas na nuvem, em comparação com ambientes dedicados. Embora a camada de virtualização ofereça um grau considerável de isolamento e segurança, é importante notar que certos elementos, como o cache e a memória principal, são compartilhados entre as várias aplicações que operam em máquinas virtuais hospedadas no mesmo servidor físico (AWAYSHEH et al., 2021; KAPIL et al., 2022).

Desde o ano de 2010, temos observado a disponibilização do serviço de nuvem co-

nhecido como "sem servidor" ou *Function as a Service* (*FaaS*), em que os clientes não se preocupam ativamente com planejamento, configuração, gerenciamento, e dimensionamento de recursos. Neste modelo de serviço, a computação é realizada em *bursts*, e seus resultados são persistidos no armazenamento definido pelo usuário. Quando uma aplicação não está sendo executada, nenhum recurso é alocado dinamicamente pelo provedor. Ao contrário do modelo *Infrastructure as a Service* (*IaaS*), em que o usuário paga pelas instâncias virtuais por segundo (*e.g.*, máquinas virtuais), independentemente se estiverem ou não em utilização enquanto instanciadas. O custo do modelo *FaaS* é de fato *pay-per-use*, pois o usuário pagará apenas pelos recursos realmente consumidos durante a execução das funções implementadas para suas aplicações. Através de uma analogia (MEDIUM, 2021) onde o usuário deseja percorrer um determinado trajeto, o modelo *IaaS* pode ser comparado ao aluguel de um carro (cujas manutenções exigidas durante o período de locação do carro são de responsabilidade do locatário), e o modelo *FaaS* pode ser comparado a utilização do *Uber*. Dentre os provedores pioneiros e populares de computação *serverless*, com alto grau de paralelismo a partir da invocação simultânea de até milhares de funções definidas em linguagens de programação suportadas pelo serviço, podemos citar *Amazon Lambda*, *Google Cloud Functions*, e *Microsoft Azure Functions*.

O desenvolvimento do modelo *MapReduce*, que se concentra nas etapas essenciais de *Map* e *Reduce*, deu origem a várias ferramentas escaláveis e confiáveis para o processamento de grandes volumes de dados. Entre os *frameworks* de código aberto disponíveis, destacam-se o *Apache Hadoop* (SHVACHKO et al., 2010) e o *Apache Spark* (ZAHARIA et al., 2012). Ambos fazem uso de um conjunto de nós computacionais para gerenciar e executar tarefas *MapReduce*. Desta forma, o modelo *IaaS* oferecido por provedores de nuvens públicas pode ser utilizado para a construção de tais *clusters* de instâncias virtuais, encontrados em, por exemplo, *Amazon Elastic Compute Cloud* (*EC2*), *Microsoft Azure Virtual Machines*, e *Google Compute Engine* (*GCE*). Além do clássico *Hadoop Distributed File System* (*HDFS*), *Hadoop* e Spark suportam diversos modos de armazenamento, como por exemplo, *Amazon Simple Storage Service* (*S3*), que é um serviço escalável, seguro e de alta disponibilidade para o armazenamento de objetos.

Mais recentemente, foi proposto MARLA (acrônimo para *MApReduce on LAmbda*) (GIMÉNEZ-ALVENTOSA et al., 2019), ferramenta que permite a configuração do modelo *serverless* para a execução de *MapReduce* no *Amazon Lambda*. O usuário implementa a lógica computacional das funções *Lambda* para cada fase do modelo *MapReduce* através dos agentes *MAPPER* e *REDUCER*, que são disparados a partir do carregamento dos dados de entrada. Atualmente, o MARLA utiliza exclusivamente o sistema de ar-

mazenamento *Amazon S3* para o armazenamento dos dados de entrada e saída de cada aplicação.

O estudo tem como objetivo, a avaliação de desempenho de uma aplicação clássica de *MapReduce* no serviço de FaaS fornecido pela AWS, em caráter exploratório de execução. Sendo os resultados obtidos comparados com a mesma aplicação executada no serviço de IaaS da AWS, mais comumente utilizado. A Aplicação escolhida é conhecida como *Word Count*, sendo esta escolhida por sua simplicidade, que consiste em ler um arquivo de texto e calcular a frequência de cada palavra nele contida. Para conduzir essa avaliação, utilizaremos o *framework* Spark em um conjunto de máquinas virtuais otimizadas para memória hospedadas no Amazon EC2, bem como o MARLA em um conjunto de funções definidas no Amazon Lambda. Ambos os ambientes utilizarão o Amazon S3 como fonte e destino dos dados. A análise abrangerá os tempos de execução e os custos financeiros associados a diversos cenários da aplicação *Word Count*. Nesses cenários, variaremos o tamanho dos dados de entrada, o número de *mappers* e *reducers*, além da quantidade de memória alocada por função no Amazon Lambda. Isso nos permitirá identificar as vantagens e limitações desse modelo de serviço na resolução desse tipo de problema, em comparação com a abordagem Spark no Amazon EC2, bem como identificação de características que levam a um otimização no tempo de execução principalmente.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 expõe o conceito do modelo *MapReduce*, mencionando e discorrendo sobre o Apache Hadoop e o Apache Spark, que são *frameworks* tradicionais que implementam essa abordagem, os conceitos básicos de nuvens computacionais, o provedor de serviços AWS, bem como alguns de seus serviços, e também conceitos de *Serverless Computing*, assim como alguns desafios e benefícios desse modelo. A Seção 3 destaca os trabalhos relacionados. A Seção 4 aborda o *framework* MARLA, descrevendo sua arquitetura e seu funcionamento em cada fase da sua execução, e também descreve as implementações da aplicação *Word Count* na *AWS*, em versões para os *frameworks* MARLA e Spark. A Seção 5 apresenta os resultados experimentais e análises das execuções da aplicação *Word Count* tanto para o MARLA quanto para o Spark, além da análise comparativa entre eles e, por fim, as conclusões e direções futuras são expostas na Seção 6.

2 Conceitos Preliminares

Este trabalho concentra esforços em identificar vantagens de executar aplicações *MapReduce* nos modelos *FaaS* e *IaaS* em nuvens. Assim, esta seção apresenta os conceitos básicos de nuvens computacionais, da nuvem pública da *Amazon* (*AWS*), e do modelo *MapReduce*, um dos mais utilizados para *Big Data*.

2.1 Modelo MapReduce

O *MapReduce* se inspira nas operações de *Map* e *Reduce* encontradas em linguagens de programação funcional, facilitando para os programadores o desenvolvimento de aplicações distribuídas (DEAN; GHEMAWAT, 2004). Algumas das características mais notáveis do MapReduce é sua capacidade de garantir tolerância a falhas e escalabilidade. Os *frameworks* atuais que implementam esse modelo lidam com a paralelização de computação, distribuição e localidade de dados, além do balanceamento de carga e da ocorrência de lentidão ou falhas em nós computacionais. Essas características permitem que o modelo seja dimensionado para *clusters* que abrangem até milhares de máquinas.

O uso de um modelo funcional com operações de mapeamento e redução, especificadas pelo usuário, permite paralelizar vários cálculos computacionais. Dado o exposto modelo *MapReduce* adota o seguinte fluxo de execução (DEAN; GHEMAWAT, 2004):

1. Os arquivos de entrada são divididos em M partes (partições, divisões ou fragmentos), onde M é definido pelo usuário.

2. Um número de cópias da aplicação é inicializado, sendo uma dessas cópias designada como Mestre (*Master*), responsável por enviar tarefas (*Tasks*) para serem processadas pelas outras cópias, denominadas Trabalhadores (*Workers*). Existem M tarefas de mapeamento e R tarefas de redução a serem atribuídas aos Trabalhadores, sendo R definido pelo usuário. O Mestre seleciona Trabalhadores ociosos e atribui a cada um deles tarefas de mapeamento ou redução.

3. Um Trabalhador que recebe uma tarefa de mapeamento (também conhecido como *Mapper*) lê o conteúdo da sua parte do arquivo de entrada, analisa-o e gera pares de chave-valor, enviando-os para a função de mapeamento (*Map*) definida pelo usuário. Os pares intermediários chave-valor produzidos pela função de mapeamento são armazenados em um *buffer* na memória.

4. Periodicamente, os pares intermediários são gravados no disco local, divididos em R regiões através de uma função de particionamento (por exemplo, hash(chave) mod R), definida pelo usuário. As localizações desses pares são enviadas de volta ao Mestre, que as encaminha para os Trabalhadores designados para tarefas de redução (*Reducers*).

5. Quando um *Reducer* é notificado sobre essas localizações, ele utiliza chamadas de procedimento remoto para ler os dados armazenados em *buffer* nos discos locais dos *Mappers*. Após a leitura de todos esses dados, o *Reducer* classifica as chaves intermediárias para agrupar todas as ocorrências por chave. Essa etapa de classificação é necessária, pois muitas vezes várias chaves diferentes são mapeadas para a mesma tarefa de redução. Se a quantidade de dados intermediários for maior do que a memória disponível, então é utilizada uma classificação externa.

6. O *Reducer* itera sobre os dados intermediários classificados e, para cada chave única encontrada, passa a chave intermediária juntamente com seu conjunto correspondente de valores para a função de redução (*Reduce*) definida pelo usuário. A saída da função de redução é anexada a um arquivo de saída final para cada partição de redução, resultando em um total de R arquivos de saída ao final da execução.

2.1.1 Apache Hadoop

A biblioteca de software denominada Apache Hadoop é uma estrutura projetada para possibilitar o processamento distribuído de grandes volumes de dados em *clusters* de computadores. Através da utilização de modelos de programação intuitivos, o Hadoop permite a escalabilidade de um único servidor para um número significativo de máquinas interconectadas, cada uma oferecendo poder de processamento e armazenamento locais. O projeto do Hadoop inclui os módulos (PANKAJ SARASWAT, 2021):

- Hadoop Common: possui utilitários que dão suporte aos outros módulos.

- Hadoop Distributed File System (HDFS™): Sistema de arquivos distribuídos utilizado pelo Hadoop.

- Hadoop YARN: Uma ferramenta para gerenciamento de tarefas e de recursos do *cluster*.
- Hadoop MapReduce: Um sistema baseado em YARN para processamento paralelo de grandes conjuntos de dados.

O Hadoop Distributed File System (HDFS) é um sistema de arquivos distribuído projetado para hardware comum. Com diferenças significativas em relação a outros sistemas do tipo, o HDFS se destaca pela sua alta tolerância a falhas e a capacidade de ser implantado em hardware de baixo custo. Ele proporciona acesso de alto desempenho aos dados de aplicativos, sendo especialmente adequado para cenários com grandes conjuntos de dados.

O Hadoop MapReduce é uma estrutura de software que permite o processamento de grandes volumes de dados de forma confiável e tolerante a falhas, em clusters grandes (milhares de nós) de hardware comum. Ele facilita a escrita de aplicativos que executam tarefas de mapeamento e redução em paralelo, seguindo uma lógica de divisão dos dados de entrada em partes independentes. A estrutura organiza e classifica as saídas dos mapeamentos, que são então utilizadas como entrada para as tarefas de redução. Tanto a entrada quanto a saída do trabalho são armazenadas em um sistema de arquivos. Além disso, a estrutura é responsável pelo agendamento das tarefas, monitoramento e reexecução das tarefas com falha. Em geral, os nós de computação e armazenamento são os mesmos, o que significa que a estrutura MapReduce e o Sistema de Arquivos Distribuídos Hadoop (HDFS) são executados nos mesmos nós.

2.1.2 Apache Spark

O Apache Spark é um poderoso *framework* de código aberto para análise de big data, permitindo o processamento de enormes volumes de dados. Com suas capacidades de computação distribuída, o Spark pode lidar de forma eficiente com bancos de dados distribuídos em um grande número de máquinas. Iniciado originalmente em 2010 na Universidade da Califórnia, Berkeley, o Spark tornou-se um projeto da Apache em 2013. Desde então, ele tem experimentado um crescimento notável e um desenvolvimento extensivo (AZEROUAL; NIKIFOROVA, 2022).

A maior vantagem do Spark é aprimorar o desempenho dos aplicativos por meio da realização de movimentações de dados em memória, sempre que possível, evitando assim custos significativos de entrada e saída de dados comumente encontrados em outros *frameworks* de *big data*, como o Apache Hadoop.

2.2 Nuvens Computacionais

Segundo Foster e Gannon (FOSTER; GANNON, 2017), a computação em nuvem é um modelo de acesso à computação e armazenamento sob demanda que fornece, através da *internet*, uma capacidade maior do que a disponível localmente. Acessar essa capacidade na nuvem pode ser mais barato, rápido e conveniente do que adquirir e operar sistemas próprios de computação e armazenamento. Os recursos podem ser acessados por meio de plataformas clientes heterogêneas (*e.g.*, estações de trabalho, *laptops* e telefones celulares) e são combinados para atender vários consumidores ao mesmo tempo, devido ao esquema *multi-tenant* utilizado pelos provedores. A gama de recursos físicos e virtuais pode ser atribuída dinamicamente para atender à demanda dos clientes, e muitas vezes parecem ilimitados em termos de tempo ou quantidade. Sua utilização é monitorada e controlada, proporcionando transparência de custos tanto para provedores quanto para clientes do serviço solicitado. Dos modos de implantação, as nuvens públicas são as mais populares, sendo *Amazon Web Services* (33%), *Microsoft Azure* (21%) e *Google Cloud* (8%) os três provedores de serviço mais utilizados do mercado *Cloud* em 2022[1].

A plataforma Amazon Web Services (AWS) é atualmente a maior provedora de serviços de computação em nuvem, com uma presença global abrangendo 99 zonas de disponibilidade em 31 regiões geográficas ao redor do mundo. A AWS oferece uma ampla gama de serviços [2], totalizando 239 categorias, incluindo o Amazon Elastic Compute Cloud (EC2), Amazon Lambda e Amazon Simple Storage Service (S3).

2.2.1 Serviços da AWS

A *Amazon Elastic Compute Cloud* (EC2) é um serviço de *Infrastructure as a Service (IaaS)* que fornece mais de 500 opções de máquinas virtuais para atender a diversos tipos de cargas de trabalho. O EC2 oferece capacidade de computação segura e escalável, eliminando a necessidade de investir em *hardware* físico. As instâncias de máquinas virtuais (MVs) são agrupadas em diferentes tipos, como *General Purpose*, *Compute Optimized*, *Memory Optimized*, *Storage Optimized* e *Accelerated Computing*. Além disso, são disponibilizadas instâncias *bare-metal*, que oferecem acesso direto ao processador e à memória do servidor para aplicativos em ambientes não virtualizados ou que implementam seu próprio *Hypervisor*. O faturamento das instâncias EC2 é feito em incrementos de segundo, hora ou mês, dependendo do tipo de instância, região e opções de preços. Existem três

[1] https://www.channele2e.com/news/cloud-market-share-amazon-aws-microsoft-azure-google/
[2] https://aws.amazon.com/pt/products/

principais opções de faturamento: *On-Demand* (sob demanda), *Reserved* (reservada) e *Spot*. As instâncias sob demanda são geralmente mais caras, mas oferecem maior estabilidade de uso. As instâncias *Spot* estão sujeitas a interrupções a qualquer momento sem duração determinada, no entanto é oferecido descontos na utilização desse modelo de faturamento.

O *Amazon Lambda* é um serviço de *Function as a Service (FaaS)* que executa código em resposta a eventos, utilizando uma infraestrutura de computação altamente disponível, seguindo o modelo "*serverless*"(sem servidor). Isso significa que os servidores são utilizados, mas o usuário não precisa se preocupar com o gerenciamento deles. Os eventos podem incluir mudanças de estado ou atualizações, como adicionar um item a um carrinho de compras em um site de comércio eletrônico.

O Lambda invoca o código do cliente quando ocorre um evento suportado pelo *AWS Lambda*, e escala automaticamente para lidar com a taxa de solicitações, que pode ser ilimitada. O cliente paga apenas pelas solicitações atendidas e pelo tempo de computação necessário para executar o código, em incrementos de um milissegundo, não sendo cobrado pela unidade do servidor que executou o código.

Atualmente, o tempo máximo de execução de uma função *Lambda* é de 15 minutos. Após esse período, a execução é encerrada. A quantidade de memória definida pelo cliente para a execução de suas funções no *Lambda* determina a alocação proporcional de capacidade de CPU, largura de banda de rede e entrada/saída de disco.

O *Amazon Simple Storage Service (S3)* é um serviço de armazenamento oferecido pela AWS e pode ser classificado como *STaaS (STorage as a Service)*. Esse serviço fornece escalabilidade, disponibilidade de dados, segurança e desempenho para armazenamento de objetos na nuvem. Os dados são armazenados como objetos em recursos denominados *buckets*, no qual um único objeto pode ocupar até 5 terabytes de espaço. Os objetos podem ser acessados por meio de pontos de acesso S3 ou pelo *hostname* do *bucket*. Esse serviço também possui recursos para mover e armazenar dados em diferentes níveis de armazenamento, configurar e aplicar controles de acesso aos dados, garantir proteção contra usuários não autorizados e executar análises de grandes volumes de dados, entre outras funcionalidades.

O serviço de monitoramento e gerenciamento conhecido como *Amazon CloudWatch* é utilizado por desenvolvedores, operadores de sistema, Engenheiros de Confiabilidade de Site e gerentes de Tecnologia da informação (IT). Ele permite a coleta de dados sobre aplicações e todo o sistema, fornecendo *insights* sobre alterações de desempenho. Esses

dados são coletados na forma de *logs*, métricas e eventos, proporcionando uma visão unificada dos recursos, aplicativos e serviços da AWS. O CloudWatch permite a definição de alarmes e a execução de ações automatizadas quando esses alarmes são acionados. É uma ferramenta valiosa, especialmente para usuários dos diversos serviços da AWS, pois permite analisar todos os serviços compatíveis em um único local, sendo compatível com mais de 70 serviços da AWS, incluindo Amazon EC2, Amazon DynamoDB, Amazon S3, Amazon ECS, AWS Lambda, Amazon EMR, entre outros, que geram métricas em intervalos de tempo definidos.

O*Amazon EMR* é um serviço que oferece uma plataforma gerenciada para processamento de grandes volumes de dados, utilizando estruturas de código aberto como Apache Spark e Apache Hadoop, utilizando instâncias do *Amazon EC2*. O *Amazon EC2* fornece máquinas virtuais com capacidade de computação, chamadas de instâncias, que são classificadas em diferentes tipos, variando em capacidade e tamanho para atender às necessidades das aplicações, como aquelas que exigem alto uso de memória.

Ao utilizar o EMR, é possível criar um cluster EMR em questão de minutos, sem a preocupação de provisionar os nós ou configurar a ferramenta necessária para a execução das tarefas (Hadoop/Spark), dentre outros. Exigindo apenas uma configuração mínima para o cluster. É possível aumentar ou reduzir o número de instâncias manualmente ou usar o *Auto Scaling*, que realiza automaticamente o redimensionamento horizontal, adicionando ou removendo instâncias de acordo com políticas pré-definidas.

2.2.2 Mercados da AWS

Em relação ao mercado de preços das instâncias EC2 da AWS, os tipos variam de acordo com a disponibilidade e tipo de serviço desejado, que irão variar de acordo com as necessidades de cada cliente, pode-se citar alguns dos principais tipos (AMAZON, 2023a):

1. **Instâncias Sob demanda (*On demand*)** são instâncias preferidas por clientes que desejam flexibilidade, sem a necessidade de pagamento inicial ou compromisso de longo prazo. Além disso, é possível ajustar a capacidade computacional de acordo com as necessidades da aplicação e das tarefas em execução. É um modelo recomendado em caso de aplicações com cargas de trabalho de curto prazo ou imprevisíveis e que não podem ser interrompidas. Neste modelo de preços, o usuário é cobrado pelo tempo em que a instância é utilizada. É essencial destacar que as instâncias sob demanda geralmente têm um custo mais alto em comparação com outros modelos

de preços.

2. **Instâncias *spot*** são recomendadas quando se tem aplicações que necessitam de um baixo custo computacional para se tornar viável, também para cargas de trabalho que são tolerantes a falhas. Em comparação com os outros modelos de precificação, este modelo oferece o maior desconto, chegando a ser até 90% mais barato que o sob demanda. No entanto, sua disponibilidade é flutuante e ocasionalmente pode apresentar volatilidade nos preços. Isso ocorre porque os clientes fazem lances em capacidade de computação ociosa, ou seja, instâncias EC2 que estão disponíveis, mas não estão em uso. Para o caso em que a instância esteja com o preço superior ao definido pelo usuário previamente, a instância não é alocada. Além disso é possível que a instância seja revogada, ou seja, seja interrompida, seja por conta de uma alta demanda de instâncias *spot*, ou pelo fato do preço *spot* ter excedido o definido pelo usuário, ou até mesmo a oferta de demandas *spot* ter diminuído.

3. **Instâncias *Reservadas* (*Reserved Instances*, RIs)** na AWS são uma opção de preço que permite a reserva de instâncias EC2 por um prazo fixo (1 ou 3 anos), em troca de uma redução na tarifa, quando comparadas com as instâncias sob demanda. Essas RIs são vantajosas para cargas de trabalho estáveis ou previsíveis, em que você pode comprometer-se com um tipo de instância específico e uma região por um período prolongado.

2.2.3 Famílias de Máquinas Virtuais da AWS

Existem várias opções de instâncias EC2 disponíveis na AWS, cada uma adequada para atender a diferentes necessidades e casos de uso específicos. Dentre os principais, encontram-se (AMAZON, 2023b):

* **Instâncias de uso Geral (*General Purpose*)**: essas instâncias são ideais para atender a diversos tipos de demandas de trabalho, pois proporcionam um equilíbrio entre recursos de computação, memória e armazenamento. Para cargas de trabalho que requerem uso intenso de recursos na execução das tarefas, sugere-se outros tipos de instâncias.

* **Instâncias Otimizadas para Computação (*Compute Optimized*)**: essas instâncias são especificamente projetadas para lidar com altas demandas de computação e se beneficiam de processadores de alto desempenho. São bem aproveitadas

em cargas de trabalho de processamento em lote, computação de alto desempenho (HPC), servidores dedicados para jogos, inferência de aprendizado de máquina, entre outros.

* **Instâncias Otimizadas para Memória (*Memory Optimized*)**: desenvolvidas com foco em memória, essas instâncias são perfeitamente adequadas para cargas de trabalho que demandam uma grande quantidade de memória, tais como bancos de dados em memória, caches e análises em tempo real. A ampla capacidade de memória permite armazenar grandes conjuntos de dados em memória, possibilitando um acesso e processamento mais rápidos dos dados.

* **Instâncias Otimizadas para Armazenamento (*Storage Optimized*)**: As instâncias otimizadas para armazenamento são projetadas para cargas de trabalho que requerem acesso rápido de leitura e gravação a conjuntos de dados muito grandes no armazenamento local, como bancos de dados de alto desempenho e *data warehouses*. Elas são otimizadas para fornecer dezenas de milhares de operações de entrada/saída (E/S).

* **As instâncias de Computação Acelerada (*Accelerated Computing*)**: projetadas para fornecer computação de alto desempenho para cargas de trabalho específicas. Elas são equipadas com aceleradores de *hardware*, como GPUs ou TPUs, que executam cálculos paralelos e aceleram tarefas relacionadas à inteligência artificial, aprendizado de máquina, processamento de dados, simulações científicas e outras cargas d e trabalho computacionalmente intensivas.

* **As instâncias Otimizadas para HPC**: são instâncias criadas especificamente para oferecer a melhor relação entre preço e *performance* na execução de *workloads* de HPC em escala na AWS. Essas instâncias são ideais para aplicações que usam processadores de alta *performance*, como simulações grandes e complexas e *workloads* de aprendizado profundo.

Essas são as famílias de instâncias disponíveis na AWS e cada uma delas inclui várias configurações específicas, permitindo que os usuários escolham a combinação certa de recursos de CPU, memória, armazenamento e desempenho para suas cargas de trabalho específicas.

Um exemplo simples pode ser observado na AWS nas famílias de instâncias otimizadas para computação no qual foram definidos padrões de nomenclatura que segue um formato

geral, servindo para diferenciar diversos tamanhos de instâncias de acordo com recursos alocados variados. A nomenclatura segue um prefixo que denota a família da instância e um sufixo que indica o tamanho específico. Por exemplo, na família "C"otimizada para computação, temos as instâncias **C5.large**, **C5.xlarge**, **C5.2xlarge** e assim por diante.

O tamanho **C5.large** oferece menos recursos de CPU e memória em comparação com o **C5.2xlarge**, que fornece maior capacidade de processamento e memória. Isso permite que os usuários escolham a instância que melhor atenda às necessidades de suas cargas de trabalho, garantindo eficiência e controle de custos. A Tabela 1 mostra a comparação dos recursos quantidade de vCPUs e quantidade de memória alocados por tamanho de instância, dentre os exemplos citados.

Tabela 1: Recursos alocados por tipo de instância.

Instance type	Default vCPUs	Memory (GiB)
c5.large	2	4.00
c5.xlarge	4	8.00
c5.2xlarge	8	16.00

2.3 Computação sem servidor (*Serverless Computing*)

A computação sem servidor é um modelo de desenvolvimento nativo da nuvem que permite aos desenvolvedores construir e executar aplicativos sem a necessidade de gerenciar servidores (MANNER, 2023).

Apesar do nome, a computação sem servidor não implica na ausência de servidores, em vez disso, significa que os desenvolvedores estão liberados das complexidades da gestão de servidores. Ao contrário dos modelos tradicionais de Infraestrutura como Serviço (IaaS), nos quais os desenvolvedores precisam lidar com a provisionamento de servidores, escalabilidade e manutenção, a computação sem servidor permite que os desenvolvedores executem código em um ambiente na nuvem sem se preocupar com esses aspectos operacionais (BALDINI et al., 2017).

2.3.1 Os Benefícios da Computação sem Servidor

No estudo (HASSAN et al., 2021), foram destacados benefícios que a computação sem servidor pode oferecer:

1. **Economia de custos:** Aplicações *serverless* são desvinculadas da infraestrutura do servidor, o que resulta em serviços com base no custo, dependendo do uso. Por

exemplo, as aplicações são executadas sempre que um usuário faz uma solicitação a um serviço dentro da aplicação. Os fornecedores de nuvem cobram apenas pelo espaço usado, e não há custo quando suas aplicações estão em estado ocioso.

2. **Escalabilidade:** A computação *serverless* resolveu de forma razoável o problema de alocação de recursos. Portanto, os desenvolvedores não precisam se preocupar com a escalabilidade da aplicação, porque a aplicação se dimensionará automaticamente sempre que as solicitações dos usuários aumentarem. Se houver inúmeras solicitações a uma função dentro da aplicação, os provedores iniciarão servidores para lidar com essas solicitações.

3. **Gerenciamento do lado do servidor:** Na computação *serverless*, os desenvolvedores não precisam se preocupar com o lado do servidor e seu gerenciamento. Os Provedores de nuvem cuidam da gestão e manutenção do *hardware* e do *software* necessários para implantar aplicações. Além disso, eles lidam com todas as operações administrativas para permitir que os desenvolvedores se concentrem em diferentes tipos de recursos, como unidade central de processamento (CPU), memória e armazenamento.

4. **Fácil implantação:** Aplicações *serverless* são fáceis de implantar. Por exemplo, para implantar uma aplicação, os desenvolvedores só precisam fazer o *upload* de algumas funções e lançar um novo produto. O servidor cuidará da gestão da implantação e das preocupações relacionadas à infraestrutura, como provisionamento de servidor e dimensionamento.

5. **Redução de latência:** Aplicações *serverless* não são hospedadas em um servidor específico; o código pode ser executado a partir de qualquer servidor em qualquer localização. Portanto, os fornecedores de nuvem podem executar a aplicação em servidores próximos à localização do usuário final. Isso reduz a latência, porque as solicitações do usuário final não precisam percorrer a Internet para acessar o servidor original.

2.3.2 Desafios da computação sem servidor

Embora a computação sem servidor ofereça inúmeros benefícios, existem estudos que evidenciam alguns desafios, e realizam considerações sobre alguns aspectos e particularidades dessa modalidade de computação. Abaixo estão alguns desafios e considerações:

1. **Partida a frio (*Cold Start*):** Um dos principais desafios da computação sem servidor é o conhecido "*cold start*". Isso se refere ao notável atraso que ocorre ao chamar uma função pela primeira vez. Esse atraso é devido à necessidade da plataforma de inicializar um contêiner que contenha todos os recursos necessários para a execução da função (KELLY et al., 2020). Métodos e técnicas para reduzir o problema da partida a frio são cruciais.

2. **Execução Prolongada:** Geralmente, os provedores de serviços em nuvem que oferecem opções de computação *serverless* estabelecem um limite de tempo máximo para a execução de uma função. Isso é feito para garantir a eficiência da alocação de recursos e evitar que funções fiquem presas em *loops* infinitos ou operações extremamente demoradas.

3. **Custo e Modelo de Preços:** O modelo de preços das soluções FaaS pode ser difícil de entender e surpreendentemente complexo para ser modelado (EIVY; WEINMAN, 2017). Além disso, há a questão do preço comparado com outros serviços oferecidos pelos provedores. No estudo conduzido em (REUTER et al., 2020), é abordado o fato da busca pela melhor opção de custo ao executar tarefas em FaaS comparando com execuções realizadas em VMaaS (*Virtual Machine as a Service*), e propões uma forma de tentar realizar as tarefas de forma híbrida, utilizando-se as duas formas.

4. **Bloqueio de Fornecedor (*Vendor lock-in*):** É um termo utilizado na computação em nuvem para descrever a situação em que os clientes ficam presos a uma única implementação tecnológica de um provedor de nuvem. Isso pode tornar difícil para um cliente mudar para outro fornecedor sem incorrer em custos significativos ou possíveis restrições legais. No contexto do FaaS, mudar de fornecedor pode ser um processo muito caro quando o ambiente de execução não é compatível com diversos fornecedores, dado que existem restrições de integrabilidade entre os fornecedores e as suas infraestruturas FaaS (GROGAN et al., 2020).

5. **Escalabilidade e Elasticidade:** Na computação sem servidor, é crucial assegurar a escalabilidade e elasticidade das funções para que possam se ajustar conforme necessário. Por exemplo, quando uma aplicação *serverless* recebe um grande volume de solicitações, é essencial que todas essas solicitações sejam tratadas. O provedor de nuvem sem servidor deve disponibilizar os recursos necessários para processar todas essas solicitações e aumentar sua capacidade à medida que o número de solicitações aumenta.

6. **Segurança:** A importância da segurança em ambientes *serverless* abrange várias áreas de preocupação, incluindo autenticação, autorização, isolamento de ambientes de execução, ataques de exaustão de recursos e questões de privacidade. A proteção contra essas ameaças é fundamental para garantir a integridade e a confiabilidade dos serviços *serverless*. Uma das questões de segurança está relacionada à isolamento, devido ao compartilhamento da plataforma por várias funções e diferentes usuários, exigindo assim, um isolamento robusto.

3 Trabalhos Relacionados

Neste capítulo, serão apresentados trabalhos que abordam execuções de aplicações com ferramentas que implementam a metodologia *MapReduce* utilizando-se do FaaS na AWS, além de trabalhos que comparam a execução de aplicações Spark na AWS.

Em seu estudo, (KIM; LIN, 2018) apresentaram o Flint, um protótipo de mecanismo de execução do Spark que oferece um modelo de custo *pay-per-use* usando o AWS Lambda. O Flint permite a execução transparente de código escrito para PySpark, a interface Python do Spark, eliminando a necessidade de configurar um cluster Spark com Máquinas Virtuais (MVs), para o processamento das tarefas. Vale ressaltar que para lidar com os dados intermediários durante o processamento, o Flint utiliza o Amazon Simple Queue Service (SQS), um serviço de mensageria fornecido pela AWS. Os autores compararam os desempenhos das execuções de aplicações no Flint em um cluster Spark com MVs na AWS EC2 (usando Python (PySpark) e Scala (*Native Spark*)), construído a partir da plataforma de análise unificada da Databricks(DATABRICKS, s.d.). Os resultados indicaram que o Flint é preferível para o processamento de análises *ad hoc* e exploratória de dados.

O trabalho desenvolvido nesta dissertação, assim como no trabalho com o Flint em (KIM; LIN, 2018), segue o caminho de análise comparativa dos tempos de execução e seus respectivos custos monetários, ao executar uma aplicação MapReduce, com os *frameworks* MARLA e o Apache Spark, agregando assim mais informações sobre execução de aplicações MapReduce com a abordagem de FaaS. A dificuldade de agregar o *framework* Flint neste conjunto de comparações está na disponibilidade e reprodutibilidade do referido *framework*.

Astrea é uma ferramenta descrita em (JARACHANTHAN et al., 2022) que visa configurar e orquestrar tarefas de análise sem servidor de forma autônoma, levando em consideração os requisitos flexíveis especificados pelo usuário. Ao formular um problema de otimização com base em requisitos específicos do usuário, Astrea busca melhorar o de-

sempenho ou reduzir custos, empregando algoritmos baseados em teoria de grafos para alcançar a execução ideal do trabalho. O trabalho apresenta experimentos representativos pois considera *benckmarks* realistas como o apresentado em (PAVLO et al., 2009). Tais experimentos consideram aplicações da área Big Data e cargas de trabalho de aplicações de aprendizado de máquina em diversas escalas.

Os benefícios do projeto e implementação através de Astrea está embasado em teoria de grafos, que oferece uma forma supostamente eficiente de execução, aprimorando custo-benefício em plataformas sem servidor. Desta forma, Astrea minimiza os desafios de configuração e orquestração de aplicações em tais ambientes.

Na abordagem do Astrea, ele roda um algoritmo para otimização baseado em requisitos especificados pelo usuário, em termos de tempo de *performance* e custo, e só depois ele fica apto para execução da aplicação em questão. Dito isto, ainda em (JARACHANTHAN et al., 2022), é mencionado que para uma configuração de um *notebook* especificada, esse algoritmo é executado em questão de segundos, e que esse tempo poderia chegar nos milissegundos se executado em um *cluster* padrão com mais capacidade computacional. Em nosso trabalho, executamos diversos experimentos com configurações diferentes, a fim de traçar análises em termos de tempo de *performance* e custos, ao realizar uma análise exploratória dos dados das execuções, buscando entender experimentalmente eventuais vantagens e desvantagens de execuções de aplicações MapReduce em relação ao *framework* Spark.

O trabalho realizado em (JONAS et al., 2017), propôs PyWren, um sistema protótipo desenvolvido em Python com AWS Lambda, que demonstra ser capaz de implementar abstrações de processamento de dados em grande escala, como o modelo MapReduce e servidores de parâmetros. Dentre algumas comparações, é descrito que em uma implementação do programa de contagem de palavras no PyWren, ao utilizar o conjunto de dados de avaliações da Amazon como entrada, obteve-se um tempo de execução de 98,6 segundos. Em contrapartida, um programa similar foi executado no PySpark com configurações semelhantes, e foi concluído em 84 segundos. Embora o PyWren tenha apresentado uma leve desaceleração devido à falta de leituras paralelas de blocos aleatórios e algumas limitações durante as operações de escrita/leitura com o S3, os resultados de tempo revelaram que o PyWren é apenas cerca de 17% mais lento que o Spark. O desempenho do PyWren mostra que é possível alcançar uma taxa de transferência alta ao acessar armazenamentos remotos, e seu uso é comparável ao desempenho de *frameworks* tradicionais, como o PySpark.

O PyWren foi desenvolvido e em seu estudo, fala-se bastante em entender a viabilidade do processamento de grande volume de dados através de abordagens como o MapReduce, por exemplo, utilizando o serviço *serverless* das nuvens computacionais. Nosso trabalho agrega a utilização de um outro *framework* que utiliza a abordagem MapReduce e o conceito *serverless*, que o compara também com ferramentas mais tradicionais e conhecidas para o processamento de BigData (Spark e Hadoop), destacando os pontos positivos e negativos dessa abordagem com *Function as a Service*.

O trabalho em (ZHANG et al., 2020), desenvolveu o *framework* Kappa. A proposta é de simplificação do desenvolvimento de aplicações *serverless*, por meio dos recursos e abstrações que o Kappa oferece para a criação de tais aplicações, onde a ideia é fazer com que o processo de desenvolvimento seja o mais semelhante possível ao desenvolvimento de uma aplicação convencional. Além disso, o Kappa utiliza *checkpointing* para lidar com *timeouts* das funções Lambda e fornece mecanismos de concorrência que permitem a computação e coordenação paralelas. Também foram executados testes de aplicações como *WordCount*, implementada com os recursos do Kappa, e comparados seus resultados com *frameworks* mais tradicionais.

Em (NUNES et al., 2021) retratou-se o provisionamento de recursos em nuvens para execução de aplicações tipo *MapReduce* com *Spark*, visando a redução de custos. A aplicação *Diff Sequences Spark* foi implementada para comparar as sequências biológicas de entrada e gerar como saída as posições onde os nucleotídeos diferem entre si. Experimentos realizados na *AWS EC2* com até 540 sequências de *SARS-CoV-2* (*COVID-19*) demonstraram o custo-benefício de instâncias otimizadas para memória, mesmo em cenários de revogação de MVs do mercado *spot*. Nos experimentos conduzidos, e exibidos na Seção 5.2, utilizamos o mesmo tipo de instância otimizada para memória.

Em (GIMÉNEZ-ALVENTOSA et al., 2019), foi desenvolvido o MARLA, acrônimo para **MA**p**R**educe on AWS **LA**mbda, e foram realizadas análises comparativas de desempenho entre o MARLA e a S*erverless Reference Architecture* for MapReduce (SRAM) da AWS. O objetivo dessas análises era avaliar os benefícios e limitações do AWS Lambda como plataforma de computação geral e para a execução de *jobs* paralelos dependentes, como aplicações MapReduce. Durante a avaliação dos resultados evidenciou-se o comportamento heterogêneo do AWS Lambda em relação à CPU e à rede, o que impacta o tempo de execução dos *jobs* MapReduce. Durante o estudo, foram identificados padrões de uso que podem mitigar essa heterogeneidade de desempenho. Por exemplo, é recomendado garantir a execução da função Lambda mesmo em máquinas virtuais com menor

capacidade computacional e considerar a possibilidade de *timeout* durante a invocação da função Lambda devido à transferência de dados em redes mais lentas, exigindo um tratamento adequado de erros no fluxo de execução da aplicação. No entanto, não foram realizadas comparações diretas com *frameworks* populares de MapReduce que utilizam a infraestrutura da AWS.

O *framework* MARLA foi escolhido para realizarmos as comparações e execuções na AWS, pois o mesmo endossava a simplicidade de seu uso, comparando-o com outros trabalhos. Além disso, foi relativamente fácil de encontrar o código fonte, e rodar a aplicação teste com o passo a passo disponibilizado pelo mesmo, enquanto que em alguns destes citados, não era possível a reprodução.

A Tabela 2, exibe as comparações entre os *frameworks* citados. A primeira coluna identifica o *framework*, seguido do seu objetivo, forma de implementação das aplicações, sua reprodutibilidade e o serviço da nuvem utilizado. Existem apenas alguns estudos que abordam o uso do *MARLA*, de acordo com o nosso conhecimento. A principal contribuição do nosso trabalho é a realização de uma comparação direta entre MARLA e *Spark* para a execução de uma aplicação MapReduce na plataforma AWS. Nosso objetivo é analisar os tempos de execução e os custos associados ao uso dos serviços *FaaS* e *IaaS*, fornecendo uma avaliação abrangente das vantagens e desvantagens em cada cenário. Além disso, tentativas de explorar as limitações atuais e suas possíveis configurações, ao escolher o serviço oferecido pela AWS (Lambda) para a execução das tarefas utilizando a abordagem MapReduce.

Tabela 2: Trabalhos relacionados.

Framework	**Objetivo**	**Implementação da aplicação**	**Reprodutibilidade**	**Serviço utilizado**
Flint (KIM; LIN, 2018)	• Execução de aplicações MapReduce ao reproduzir ambiente Spark utilizando funções Lambda	• Implementação utilizando PySpark similar à implementação para o cluster Spark	• Não foi possível encontrar forma para reproduzir os experimentos	AWS Lambda

(Continued on the next page)

Tabela 2: Trabalhos relacionados (cont.)

Framework	**Objetivo**	**Implementação da aplicação**	**Reprodutibilidade**	**Serviço utilizado**
Astrea (JARACHANTHAN et al., 2022)	• Otimizar e executar aplicações MapReduce utilizando funções Lambda, dadas restrições impostas pelo usuário (custo de execução e tempo de execução)	• Não foi mencionado a forma de implementação das aplicações com o framework em questão	• Não foi possível encontrar forma para reproduzir os experimentos	AWS Lambda
PyWren (JONAS et al., 2017)	• Execução de aplicações MapReduce utilizando notações específicas do framework durante implementação da aplicação	• Utilização de funções específicas do framework em código python, que permite, por exemplo, iniciar a execução das funções Lambda	• Passo a passo para execuções de testes pode ser encontrado no GitHub	AWS Lambda
MARLA (GIMÉNEZ-ALVENTOSA et al., 2019)	• Execução de aplicações MapReduce via carregamento de um arquivo alvo em um bucket S3 especificado	• Implementação de duas funções em python (mapper e reducer)	• Passo a passo para execuções de testes pode ser encontrado no GitHub	AWS Lambda
Kappa (ZHANG et al., 2020)	• Simplificação do desenvolvimento de aplicações serverless, escrita em python, com modificações específicas para o framework	• Notações específicas no código python, como spawn() e checkpoint(), indicando a inicialização de uma função Lambda e a persistência do estado da execução, respectivamente	• Passo a passo para execuções de testes pode ser encontrado no GitHub	AWS Lambda
Spark (NUNES et al., 2021)	• Execução de aplicações MapReduce utilizando cluster construído com instâncias da AWS ao rodar jobs no nó Master	• Implementação utilizando estruturas de dados e funções específicas do Spark em python	• Código para criação do cluster spark e submissão dos jobs pode ser encontrado no GitHub	AWS EC2

4 O Ambiente de análise

Este capítulo descreve o framework MARLA (GIMÉNEZ-ALVENTOSA et al., 2019), abrangendo sua arquitetura, funcionamento e características principais. Além disso, é descrito como foram as implementações da aplicação WordCount, utilizando o MARLA e também o Spark (APACHE, 2022).

4.1 Arquitetura do MARLA

A arquitetura de MARLA é composta por serviços disponíveis na nuvem, mais especificamente aqueles fornecidos pela AWS (*Amazon Web Services*). MARLA é fundamentada em três tipos de funções Lambda: O *COORDINATOR*, o *MAPPER* e o *REDUCER*, além de utilizar dois *buckets* do AWS S3, sendo um para os dados de entrada, o *Input Bucket*, e o outro para armazenamento dos dados processados, *Output Bucket*.

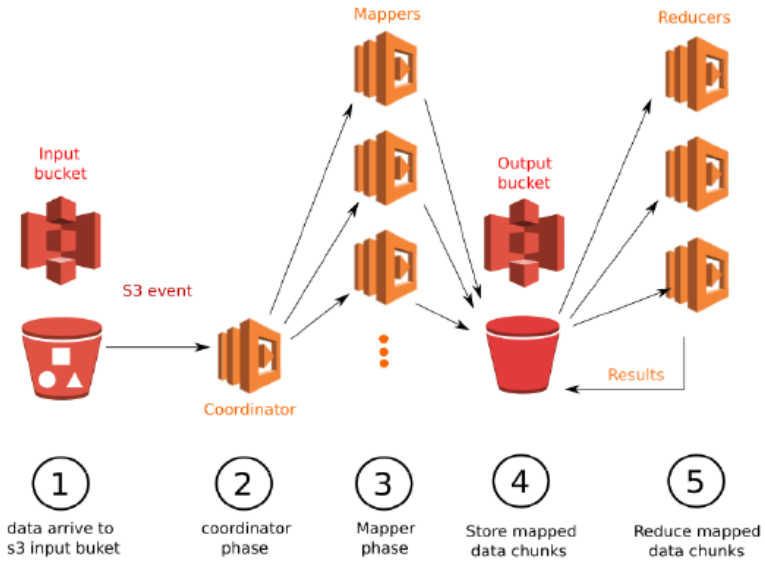

Figura 1: *Arquitetura do MARLA na AWS*

Quando um arquivo texto é carregado no *Input Bucket*, é iniciado o fluxo de execução de trabalho MapReduce. A partir daí, um evento é acionado para a função Lambda

COORDINATOR. Essa função, por sua vez, calcula o tamanho ideal dos pedaços de dados em que o arquivo será dividido, levando em consideração o número de pedaços indicado pelo usuário em um arquivo de configuração, e a capacidade de armazenamento da memória alocada para cada função *MAPPER*. Em seguida, cada *mapper* faz o download de sua parte do arquivo de entrada para executar a fase de mapeamento em paralelo com os outros *mappers*, produzindo uma lista de pares de chave/valor que serão reduzidos na próxima fase (GIMÉNEZ-ALVENTOSA et al., 2019) à solução desejada.

4.1.1 COORDINATOR

A função Lambda *COORDINATOR* é responsável pela divisão do arquivo texto de entrada em pedaços (referenciados aqui por *chunks*), que serão processados posteriormente pela função Lambda *MAPPER*. A divisão ocorre levando em conta o tamanho do arquivo de entrada e a quantidade de *mappers*, quantidade esta que é definida pelo usuário em um arquivo de configuração, que será explicado na Seção 4.2. O tamanho do *chunk* a ser processado por cada *mapper*, denotado por *chunksize*, foi especificado em (GIMÉNEZ-ALVENTOSA et al., 2019) da seguinte forma:

$$chuncksize = \frac{totalDataSize}{N_{mappers}} \tag{4.1}$$

onde $totalDataSize$ é o tamanho do arquivo de entrada e $N_{mappers}$ é o número de *mappers*, informações estas fornecidas pelo usuário.

Com o *chunksize* calculado, o *COORDINATOR* executa algumas validações de acordo com mais dois parâmetros especificados pelo usuário no arquivo de configuração explicado na Seção 4.2, são eles o *MINBLOCKSIZE* e o *MAXBLOCKSIZE*, que identificam respectivamente o menor e o maior valor que o *chunksize* pode assumir, calculado conforme Equação 4.1. Essas validações, funcionam basicamente para adequar o valor do *chunksize* calculado com os parâmetros de configuração definidos pelo usuário. Além disso, com os parâmetros informados, ou até mesmo por uma questão de memória da função Lambda, como veremos a seguir, o próprio *COORDINATOR* pode definir um novo valor para *chunksize*, e recalcular o novo número de *mappers* com base no novo *chunksize* definido. O *COORDINATOR* calcula um valor chamado de *safeMemorySize*, que indica o tamanho máximo da entrada que cada função Lambda *mapper* poderá processar, esse valor é calculado a partir de uma porcentagem da memória definida para a função Lambda *MAPPER*. Após a validação e definição do número de *mappers* e o *chunksize*, o *coordinator* chama o primeiro *mapper*, dando início a fase *MAPPER*. Os *mapper*s restantes serão invocados a

partir desta primeira invocação, conforme apresentada na subseção a seguir.

As validações ocorrem na seguinte ordem(GIMÉNEZ-ALVENTOSA et al., 2019):

1. Se o *chunkSize* for menor que o tamanho mínimo de bloco *MINBLOCKSIZE*, o *chunkSize* é ajustado para *MINBLOCKSIZE* e o $N_{mappers}$ é recalculado usando a fórmula:
$$N_{mappers} = int(\frac{totalDataSize}{MINBLOCKSIZE}) + 1 \tag{4.2}$$

2. Se o *chunkSize* exceder o tamanho seguro de memória calculado ($safeMemorySize$), o qual é determinado como uma fração da memória alocada para as funções de *mapper* pelo coordenador, o *chunkSize* é definido como $safeMemorySize$. O $N_{mappers}$ é então recalculado usando a fórmula:
$$N_{mappers} = int(\frac{totalDataSize}{safeMemorySize}) + 1 \tag{4.3}$$

3. Se o *chunkSize* for maior que o tamanho máximo de bloco *MAXBLOCKSIZE*, o *chunkSize* é ajustado para corresponder a *MAXBLOCKSIZE* e o $N_{mappers}$ é recalculado usando a fórmula:
$$N_{mappers} = int(\frac{totalDataSize}{MAXBLOCKSIZE}) + 1 \tag{4.4}$$

É importante notar que em todas as redefinições anteriores do número de *mappers* $N_{mappers}$, a função *COORDINATOR* sempre inclui um *mapper* adicional encarregado de processar o fragmento residual de dados com o seguinte tamanho:

$$residualData = totalDataSize - (Nmappers - 1) \times chunkSize \tag{4.5}$$

Vale ressaltar que o tamanho dos dados residuais, no pior cenário, será de *chunkSize* - 1 bytes. Esse método evita que qualquer *mapper* processe seu fragmento correspondente, juntamente com os dados residuais mencionados, o que poderia causar uma sobrecarga adicional a este *mapper*. Todavia, os dados residuais podem ser pequenos o suficiente para criar um *mapper* subutilizado que conclui sua execução mais rapidamente em comparação com os outros *mappers*.

4.1.2 MAPPER

A função Lambda *MAPPER* inicia com uma estratégia de redução logarítmica para invocar os *mapper*s restantes. Como resultado, cada invocação da função Lambda será responsável por invocar outra, em uma série de iterações, onde em cada iteração, os *mappers* existentes invocam mais um *mapper*, até que se atinja a quantidade de *mapper*s definida previamente (pelo usuário ou alterada pelo *COORDINATOR* após as validações).

Após a finalização da fase de invocação dos *mapper*s, é iniciado o processo de mapeamento:

- **Download e Mapeamento:** Cada *mapper* faz o download de seu *chunk* de dados. A seguir, a função de mapeamento definida pelo usuário é aplicada, sendo o resultado uma lista de pares chave/valor.
- **Preparação para a Fase de Redução:** Uma função Lambda do *MAPPER* ordena os dados. Os resultados mapeados são divididos em fragmentos (*chunks* mapeados), sendo cada um processado por uma função Lambda de independente na fase de redução.
  - **Divisão dos Dados para fase de Redução:** O agrupamento dos fragmentos é baseado no primeiro caractere da chave de cada par de chave/valor. Os fragmentos são gerados de forma a serem independentes de acordo com uma relação de ordenação. Isso garante que pares com a mesma chave sejam processados juntos na fase de Redução.
  - **Armazenamento dos Fragmentos:** Os fragmentos são armazenados no *bucket* de saída. Cada fragmento é identificado pelo número do *reducer* que o processará e o *mapper* responsável pelo mapeamento do fragmento. É adicionado um prefixo a cada fragmento, mais especificamente o prefixo `hash md5` que garante uma distribuição eficiente dos fragmentos no `S3`, melhorando o desempenho de acesso aos dados armazenados no `S3`.
- **Função de Teste Final:** Antes do término da execução da função Lambda do *MAPPER* da penúltima partição, é realizada uma invocação da primeira função Lambda de *REDUCER*, que executa um teste conhecido como *Tester Function*.

Apesar de *Tester Function* ter sido iniciado, não é garantido que a fase *MAPPER* tenha sido finalizada, uma vez que a responsabilidade dessa função é verificar que todas

as partições mapeadas correspondentes tenham sido carregadas, verificando os fragmentos enviados ao S3.

Em seguida, *Tester Function* prosseguirá para invocar todos os *reducers*, definidos pelo usuário, antes de finalmente concluir sua execução e dar início a fase do *REDUCER*.

4.1.3 REDUCER

Na fase de Redução, todos os *reducers*, iniciados pelo *Tester Function*, aplicarão as seguintes tarefas, conforme listadas a seguir:

- **Carga e Redução:** Os fragmentos que foram armazenados no *bucket* de saída no S3, serão recuperados e carregados considerando o tamanho da memória escolhido pelo usuário. Para validar o tamanho de memória, para cada carga de um fragmento que foi gerado por um *mapper*, é realizada uma validação que compara o tamanho do fragmento com a quantidade de memória definida na função *reducer*, o que não pode ultrapassar 3%. Esse valor é definido internamente por MARLA. Caso ultrapasse esse valor durante o processamento, a função é terminada.
- **Processamento de Todos os Fragmentos:** O ciclo de carga e processamento é repetido até que todos os fragmentos correspondentes tenham sido processados. A cada carga de novos fragmentos, todos são processados juntamente com os resultados dos fragmentos anteriores.
- **Armazenamento dos Resultados:** Os resultados são armazenados no *Output Bucket*. Cada função de Redução produzirá um arquivo de resultados com seu identificador.

O MARLA não realiza automaticamente a agregação dos resultados dos *reducers*. O usuário é responsável por realizar a agregação dos resultados finais, se necessário, em um pós-processamento adicional, visto que o resultado final do processamento com o MARLA possui comportamento similar ao do *framework* Apache Hadoop, o qual vários *reducers* geram diferentes arquivos de saída.

4.2 MARLA - Considerações e Configurações

Para o correto funcionamento do MARLA, são necessárias algumas configurações antes da criação das funções Lambda na AWS. Além disso, é fundamental abordar algumas

características inerentes ao funcionamento deste *framework*.

4.2.1 Arquivo de configuração

Para o MARLA criar as funções Lambda e configurar toda a aplicação na AWS é necessário que o usuário, além de, definir as funções de mapeamento e de redução, preencha também parâmetros em um arquivo de configuração. A seguir serão apresentados os possíveis parâmetros nesse arquivo:

1. **ClusterName:** Identificador para o "cluster Lambda";
2. **FunctionsDir:** Diretório que contém o arquivo que define as funções *MAPPER* e *REDUCER*;
3. **FunctionsFile:** O nome do arquivo com as funções *mapper* e *reducer*;
4. **Region:** A região da AWS onde as funções AWS Lambda serão criadas;
5. **BucketIn:** O nome do *bucket* no S3 para os arquivos de entrada, que deve ser criado previamente;
6. **BucketOut:** O nome do *bucket* no S3 para os arquivos de saída. É recomendado que sejam *buckets* diferentes para entrada e saída, a fim de evitar recursões indesejadas;
7. **RoleARN:** O ARN (Amazon Resource Names) do recurso que possui armazenado as permissões necessárias para executar a função Lambda. É necessária a permissão para deletar, criar e listar objetos no S3, além de permissão para invocar funções Lambda.
8. **MapperNodes:** O número desejado de *Mappers* a serem executados em paralelo;
9. **MinBlockSize:** O tamanho mínimo, em KB, do texto que cada *mapper* irá processar;
10. **MaxBlockSize:** O tamanho máximo, em KB, do texto que cada *mapper* irá processar;
11. **MapperMemory:** A memória (em MB) das funções Lambda *MAPPER*. O tamanho máximo do texto a ser processado por cada *mapper* será limitado por essa quantidade de memória;

12. **ReducerMemory:** A memória (em MB) das funções Lambda *REDUCER*;

13. **TimeOut:** O tempo limite (em segundos) para a execução de uma função Lambda. Após esse tempo a função será terminada.

14. **ReducersNumber:** O número de *reducers* a serem utilizados.

4.2.2 Tratamento de falhas

Quando ocorrem exceções devido a uma entrada inválida do usuário, a invocação da função Lambda é encerrada, interrompendo assim o processamento de dados. É de conhecimento também (AMAZON, 2023c), que as funções Lambda possuem um número máximo de vezes que podem ser reexecutadas em caso de falha, podendo este valor ser alterado. Da mesma forma, se forem detectadas falhas durante os processos de armazenamento ou de carregamento dos dados do S3, as etapas de processamento atuais ou subsequentes não conseguirão encontrar os dados necessários, encerrando a execução das funções Lambda. No estudo (GIMÉNEZ-ALVENTOSA et al., 2019) é citado que, dependendo do momento em que a falha ocorra, os comportamentos serão diferentes:

- **Falha no *COORDINATOR*:** Nenhum *mapper* será invocado e a execução será encerrada;

- **Falha no *MAPPER*:** Uma vez que o *mapper* falha, o armazenamento dos dados no S3 não ocorre, dessa forma o *Tester Function* não encontrará todos os fragmentos e não dará início a fase de redução. Por outro lado, os *mappers* que executarem com sucesso terão seus dados mapeados no *bucket* S3 de saída com sucesso;

- **Falha no *REDUCER*:** De forma similar, caso a falha ocorra em um *reducer*, afetará apenas ele mesmo, enquanto que os demais *reducers* caso executem com sucesso, terão seus resultados armazenados com sucesso no *bucket* S3.

Fica evidente que para qualquer falha em uma das fases, a execução será encerrada ou ficará incompleta, indicando que não há um tratamento adequado para falhas no MARLA. É de conhecimento que aplicações que guardam estado para se recuperar de uma falha, ou que implementam algum outro mecanismo para se recuperar de uma falha, de certa forma adicionam um custo computacional para realizar tal ação, o que no caso do MARLA não tem.

4.2.3 Considerações e Limitações

É importante destacar ainda, as considerações e limitações mencionadas em (GIMÉNEZ-ALVENTOSA et al., 2019):

- a versão atual de MARLA funciona com arquivos de dados em formato de texto simples, baseados em colunas, onde a mudança de coluna é especificada pelo caractere ",";
- Os dados de entrada para processamento não precisam ser pré-particionados. O MARLA realizará automaticamente a etapa de particionamento de dados considerando a configuração especificada pelo usuário e a memória alocada para as funções;
- As funções de "mapeamento"e "redução"do usuário devem ser implementadas em Python, que é a única linguagem suportada no momento;
- As credenciais de acesso e as funções IAM necessárias para usar os serviços da AWS devem ser fornecidas pelo usuário.

4.3 Implementação da Aplicação *Word Count*

Como representante da classe de aplicações MapReduce, foi escolhido a aplicação *Word Count*, clássica no domínio *MapReduce*, que contabiliza o número de ocorrências de cada palavra única, dado um conjunto de arquivos de entrada especificado pelo usuário.

4.3.1 MARLA - Implementação

No *framework* MARLA, a implementação da aplicação é simplificada ao exigir apenas a codificação de duas funções em Python: a função do mapeamento (*mapper*) e a função de redução (*reducer*). Sendo assim, a implementação seguiu da seguinte forma:

- ***Função mapper***: A assinatura da função *mapper* é predefinida pelo MARLA, na qual a entrada é um dos fragmentos de dados que foram divididos na etapa do *COORDINATOR*, ou seja, o argumento de entrada é um texto, que na linguagem python esta representado como String, e sua saída é uma lista de pares *chave-valor* cujas chaves são palavras, e os valores são números inteiros que representam suas respectivas frequências. A implementação contém os seguintes passos:

1. Leitura do fragmento de texto onde este é processado e transformado em uma lista de palavras. Esse processamento é realizado utilizando a função *split* disponível na linguagem Python, que divide o texto com base em um delimitador, que no caso abordado neste trabalho, são espaços em branco, gerando assim uma lista contendo cada palavra individualmente.
2. Geração dos pares $\langle chave, valor \rangle$, percorrendo a lista de palavras produzida no passo anterior e para cada palavra, é gerado um par onde a chave é esta palavra e o valor é 1, que é o número de ocorrência. Vale ressaltar que todas as ocorrências de vírgula (que é um caractere especial para o processamento de dados parciais no *MARLA*) nas palavras são removidas, e além disso também são descartadas palavras vazias.

- **Função *reducer***: Da mesma forma que a função *mapper* tem sua assinatura predefinida, a função *reducer* também. A entrada da função são os pares de $\langle chave,valor \rangle$ produzidos pela função *mapper* e a saída também uma lista de pares $\langle chave,valor \rangle$, porém com suas chaves agrupadas e os valores (ocorrências) somados, ou seja, na saída, palavras iguais são exibidas apenas uma vez com o valor total da quantidade de vezes que ela apareceu. É importante ressaltar que o MARLA ordena a lista produzida no *mapper*, com isso:

  1. O *reducer* itera sobre a lista de entrada, adicionando o primeiro par $\langle chave,valor \rangle$ na lista de saída, verificando as chaves de cada par subsequentes.
  2. Caso a chave seja diferente da anterior, adiciona-se esse novo par chave-valor na lista.
  3. Caso a chave seja igual a anterior, incrementa-se o valor do par na lista de saída com o valor referente ao par atual.

4.3.2 Spark - Implementação

O modelo *MapReduce* pode ser expresso no *framework Spark* através das seguintes operações de transformação: i) *flatMap* para a fase *Map*; e ii) *groupByKey* ou *reduceByKey* para a fase *Reduce* (ZAHARIA et al., 2012). Como todas as transformações no *Spark* seguem a técnica de programação funcional *Lazy Evaluation*, que adia a computação até que um determinado ponto de execução exija os seus resultados, é essencial utilizar pelo menos uma operação de ação (*e.g.*, *collect*, *saveAsTextFile*) para ativar a computação dos dados.

Em relação às estruturas de dados oferecidas pelo *Spark*, que são coleções imutáveis de objetos de dados distribuídos logicamente entre os nós computacionais (*Workers*) para processamento paralelo, foi adotado o *RDD* (*Resilient Distributed Dataset*). A execução do *Word Count* no *Spark* consiste das seguintes etapas:

1. Leitura do arquivo de entrada através da função *textFile*, gerando o RDD_1, cujos elementos são blocos de texto;

2. Definição do número de partições (*map tasks*) para o RDD_1 através das funções *coalesce* ou *repartition*, gerando o RDD_2;

3. Transformação do RDD_2 através da função *flatMap*, que fragmentará cada bloco de texto (elemento) em palavras, gerando o RDD_3;

4. Transformação do RDD_3 através da função *Map*, que mapeará cada elemento (palavra) em um par *chave-valor*, gerando o RDD_4;

5. Definição do número de partições (*reduce tasks*) para o RDD_4 através das funções *coalesce* ou *repartition*, gerando o RDD_5;

6. Transformação do RDD_5 através da função *reduceByKey*, que agrupará os pares *chave-valor* por chave (palavra) e somará todos os respectivos valores (frequências), gerando o RDD_6;

7. Gravação do RDD_6 em arquivos de texto, um arquivo por partição de redução, através da ação *saveAsTextFile*.

O capítulo detalhou as funcionalidades e responsabilidades específicas das funções Lambda dentro do MARLA, oferecendo um entendimento mais claro de como o *framework* gerencia tarefas de processamento de dados em um ambiente de nuvem. A implementação da aplicação *WordCount* usando MARLA e Spark é discutida, fornecendo detalhes da implementação.

5 Resultados Experimentais na AWS

Este capítulo tem como objetivo analisar experimentalmente a execução de aplicações MapReduce sob o paradigma FaaS, utilizando o serviço AWS Lambda, no qual pretende-se identificar as vantagens e desvantagens dessa abordagem. Para atingir esse objetivo, foi utilizado o *framework* *MARLA* para executar a aplicação *WordCount*. Além disso, realizamos uma análise comparativa entre a execução da aplicação em ambiente FaaS e em ambiente IaaS, sendo neste utilizado o *framework* *Spark* no AWS EC2.

O *Amazon S3* foi utilizado para o armazenamento dos dados de entrada, intermediários (apenas para a versão com *MARLA*) e de saída. Foram coletados os tempos (em segundos) e custos monetários (em *USD*) de execução em diversos cenários para cada implementação. No caso da implementação com o *MARLA*, foi utilizado o serviço *CloudWatch* que retorna as estimativas de interesse. Já na versão *Spark*, a implementação foi instrumentada para que tais estimativas fossem coletadas, visto que haveria dificuldade por parte do *CloudWatch* de resgatá-las, considerando as tarefas executadas internamente a um cluster *Spark*.

O MARLA foi projetado para executar as tarefas recebendo, como entrada, um arquivo baseado em colunas, conforme constatado na Subseção 4.2.3. Para a execução do WordCount, foi elaborado um pequeno tratamento para que fosse possível realizar adequadamente a contagem das palavras em textos comuns da base do projeto Gutenberg (PROJECT..., 2023). Os arquivos de entrada para as aplicações foram constituídos da seguinte forma:

1. Arquivo de 2GB foi formado a partir do download dos textos do projeto mencionado.

2. Arquivo de 4GB foi formado pela concatenação do arquivo de 2GB com ele mesmo, a partir do comando "cat"do linux, executado via linha de comando. Exemplo: »cat arquivo-2GB.txt arquivo-2GB.txt > arquivo-4GB.txt". Dessa forma, foi concatenado o arquivo 2GB com ele mesmo formando o de 4GB.

3. Arquivo de 10GB foi formado igualmente ao de 4GB, porém o arquivo base de 2GB foi diferente. Exemplo: »cat arquivo-2GB.txt arquivo-2GB.txt arquivo-2GB.txt arquivo-2GB.txt arquivo-2GB.txt > arquivo-10GB.txt".

5.1 Análise dos Parâmetros de Execução Configuráveis

A escolha dos parâmetros para uma boa configuração e consequente bom custo/benefício, na maioria das vezes, é uma tarefa não trivial. Nesta seção serão analisados os parâmetros que podem ser configuráveis dentro do ambiente Lambda e da aplicação MapReduce:

- quantidade de memória;
- número de *mappers*;
- número de *reducers*;

Tais especificações condicionam a execução da abordagem *MapReduce* utilizando as funções Lambda da AWS. É essencial compreender e otimizar os parâmetros de configuração, pois irão influenciar diretamente na eficiência, desempenho e custo do processamento distribuído.

Com relação ao número de *reducers*, o próprio trabalho onde foi desenvolvido o MARLA(GIMÉNEZ-ALVENTOSA et al., 2019), comenta que esse número, é interessante que esteja na casa das dezenas. E o que podemos reparar, é que até 128 *reducers* configurados para a execução da aplicação, obteve-se ainda um ganho de desempenho na maioria das execuções.

Ao iniciar os testes com a implementação do *Word-Count* no MARLA, um dos primeiros desafios foi determinar a quantidade de memória necessária para evitar que a função *Lambda Mapper* terminasse devido a um "estouro"de memória. O término prematuro durante a execução de uma etapa interna pode ocorrer devido a impossibilidade de processar seu *chunk* de dados associados por falta de memória. Como resultado, a execução não avança para a fase de redução.

Para tal estudo, foram executados uma sequência de testes a fim de encontrar o menor número de *mappers* capaz de processar o arquivo de entrada, dada a memória configurada da função *Lambda Mapper*. Tanto a quantidade de *mappers* quanto de memória são parâmetros que podem ser definidos no arquivo de configuração do MARLA, apresentado na Subseção 4.2.1. Os testes foram executados utilizando as quantidades de memória #

Mem = {1024MB, 2048MB, 4096MB, 8192MB} e variando-se a quantidade de *mappers* # **Map** utilizando uma abordagem de busca binária.

Inicialmente, a quantidade de memória foi configurada como # **Mem** = 1024MB, e o menor número de *mappers* que não causasse estouro de memória foi derivado através de uma busca binária dentro do intervalo [1,100]. Como resultado, chegou-se a # **Map** = 58. Para os outros tamanhos de memória, este percentual foi aplicado para calcular o tamanho do *chunk* em relação à quantidade total de memória alocada para a função, e consequentemente, o número de *mappers* derivado. Uma pequena busca linear (decrementos de 1) ainda foi realizada para se chegar à quantidade mínima de *mappers* nas outras configurações de memória (2048MB, 4096MB e 8192MB).

Como resultado, a tabela 3 mostra o número mínimo de *mappers* que cada configuração de memória definida necessita para que não ocorra o "estouro"de memória. O *chunksize* (em bytes), foi coletado através de informações que são geradas na fase do *Coordinator*, pois este é o responsável por definir o tamanho que cada *mapper* irá processar. Todos os testes utilizaram como base um arquivo texto de 2GB de entrada.

Tabela 3: Quantidade mínima de mappers por memória definida.

Arquivo de entrada 2GB			
LFMem (MB)	**Chunksize (bytes)**	**Nº mínimo de mappers**	**Chunksize / memória (%)**
1024	35,310,344	58	3.45
2048	75,851,851	27	3.70
4096	146,285,714	14	3.57
8192	292,571,428	7	3.57

Na Tabela 3, para cada *Lambda Function Memory* (LFMem), o número mínimo de mappers foi identificado juntamente com o seu *ChunckSize*. A última coluna da tabela apresenta o percentual $\frac{ChuckSize}{LFMem}$ que retrata o percentual do *chunksize*, que cada função Lambda foi capaz de processar variou de 3,45% até 3,70% do total de memória definido para a execução em questão. Portanto, isso significa que a maior quantidade de dados de texto para serem processados em cada função *Lambda Mapper*, é no melhor dos casos 3,70% da memória alocada para a função, para os cenários executados nesse experimento.

Dito isto, em relação ao alto consumo, foi verificado através de métricas que coletavam a quantidade máxima de memória utilizada, logo após cada etapa interna da função *Lambda Mapper*, e foi constatado que o maior uso da memória na fase de mapeamento, é alcançado durante uma etapa que ordena os pares de chave/valor, etapa esta que é orquestrada pelo *framework* MARLA.

Dessa forma, é crucial evitar o "estouro de memória"ao trabalhar com funções Lambda, pois quando uma função excede o limite de memória definido, sua execução é interrompida. Além da interrupção da execução, observou-se que alguns *mappers* que utilizavam mais memória do que o definido para a função continuavam em execução até atingirem o tempo limite que foi estabelecido no arquivo de configuração do MARLA. Como resultado, além de falharem, todo o tempo em que esses *mappers* executaram, será cobrado pelo provedor, no nosso caso a AWS, gerando custos desnecessários.

Por fim, esperava-se que o *Coordinator* recalculasse o número de *mappers* de acordo com o valor chamado de *safeMemorySize*, conforme explicado na Subseção 4.1.1. Esse cálculo deveria ocorrer durante a execução para garantir um ajuste dinâmico dos recursos, para evitar o encerramento inesperado da execução por utilização da memória acima da definida. Porém esse valor está definido como 30% da memória total determinada para a função Lambda, em contraste com os 3,45% - 3,70% que encontramos executando os experimentos. A diferença observada provavelmente se deve à natureza dos experimentos conduzidos no trabalho (GIMÉNEZ-ALVENTOSA et al., 2019), onde o MARLA foi desenvolvido. Esses experimentos consistiram em tarefas de MapReduce que executavam *queries* para seleção de alguns valores em um arquivo de texto, baseado em coluna. Portanto, essas tarefas não alcançaram a mesma quantidade de pares de chave/valor gerados ao contabilizar a frequência de todas as palavras da entrada na aplicação *WordCount*.

No *WordCount*, o processamento envolve uma fase de mapeamento que gera pares de chave/valor para cada palavra da entrada, seguida de uma fase de redução que contabiliza a frequência de cada palavra. Como resultado, o número de pares de chave/valor gerados pode ser muito maior em comparação com as tarefas de *queries* que foi utilizada em (GIMÉNEZ-ALVENTOSA et al., 2019). Essa diferença na geração de pares de chave/valor pode explicar a discrepância nos valores de consumo de memória entre os experimentos realizados e a aplicação *WordCount*.

Portanto, é fundamental encontrar uma configuração adequada para os parâmetros de memória e *mappers* que permita o processamento eficiente da aplicação sem exceder os recursos disponíveis e, assim, garantir um funcionamento adequado e econômico no ambiente de execução das funções Lambda.

5.2 Análise dos Resultados com MARLA

Para analisar o *Word-Count-MARLA*, foram especificadas diferentes capacidades de memória, representadas por # **Mem**, com valores de 1024 MB, 2048 MB, 4096 MB e 8192 MB por função Lambda, com o objetivo de abranger uma ampla variação em relação ao mínimo, que é 128 MB, e o máximo, que é 10240 MB. As quantidades de *mappers* e *reducers*, denotadas por # **Map** e # **Red**, respectivamente, foram variadas considerando as seguintes características:

i) # **Map** – número de *mappers* que foi variado em potências de 2 até o máximo de 128. O valor mínimo da # **Map** respeita o necessário para execução da fase *Map* no *MARLA*. Essa abordagem foi utilizada de modo a evitar estouros de memória, uma vez que a etapa *Map* demanda um alto consumo de memória, o qual será discutido mais a frente na Seção 5.1.

ii) # **Red** – número de *reducers*, que foi variado de 4 até 128, em potências de 2, com a garantia que não excedesse o número correspondente de *mappers* (# **Map**) em cada cenário.

Para cada combinação de valores, os volumes de dados de entrada (I) considerados foram 2 GB e 4 GB.

5.2.1 Execuções do *Word-Count-MARLA* com entrada de 2GB de dados

A **Tabela 4** apresenta as médias dos tempos de execução e respectivos custos monetários para a entrada de 2 *GB*, considerando cinco execuções por cenário. Foram também coletados os tempos associados aos agentes de cada fase no *MARLA*, *i.e.*, os tempos do *COORDINATOR* (responsável por realizar a divisão e distribuição dos dados de entrada), e os tempos das fases *MAPPER* e *REDUCER*, separadamente.

As funções Lambda com # **Mem** = 1024 *MB* apresentaram tempos significativamente maiores, em torno de 73%, quando comparadas aos cenários com # **Mem** = {2048, 4096, 8192} *MB*. Isto deve-se ao fato de que o Lambda aloca poder computacional (quantidade de vCPUs) proporcional à quantidade de memória provisionada, sendo 1 vCPU para 1769 MB e 6 vCPUs com 10240 MB (AMAZON, 2020, 2023d) . As funções com # **Mem** =

Tabela 4: 2GB - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 2 GB Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
1024	**64**	**4**	0.96	49.47	84.58	135.01	0.06
		8	1.00	50.47	71.30	122.77	0.06
		16	1.00	49.73	38.23	88.96	0.06
		32	0.99	51.17	30.16	82.32	0.07
		64	0.95	52.03	22.90	**75.88**	0.08
	128	**4**	1.42	24.88	82.35	108.65	0.06
		8	1.28	25.41	83.30	109.99	0.07
		16	0.96	25.30	44.74	71.00	0.07
		32	0.97	25.93	36.49	63.39	0.07
		64	0.98	27.68	28.77	57.43	0.09
		128	0.97	29.77	21.10	**51.84**	0.11
2048	**32**	**4**	0.92	53.54	36.65	91.11	0.06
		8	0.98	53.46	32.61	87.05	0.07
		16	0.97	54.48	20.65	76.10	0.07
		32	0.97	54.69	16.67	**72.33**	0.08
	64	**4**	0.98	28.02	42.60	71.60	0.07
		8	1.02	28.33	39.57	68.92	0.07
		16	1.12	28.71	24.91	54.74	0.07
		32	1.06	29.40	20.27	50.73	0.08
		64	0.93	31.04	15.67	**47.64**	0.10
	128	**4**	0.99	14.51	51.90	67.40	0.07
		8	0.95	14.62	44.87	60.44	0.07
		16	0.94	14.84	29.66	45.44	0.08
		32	0.99	15.24	24.51	40.74	0.09
		64	0.95	16.22	20.82	37.99	0.11
		128	1.03	17.51	15.31	**33.85**	0.14
4096	**16**	**4**	1.23	102.31	29.48	133.02	0.12
		8	1.25	104.27	26.57	132.09	0.13
		16	1.22	105.80	16.08	**123.10**	0.13
	32	**4**	1.01	53.17	38.44	92.62	0.12
		8	1.00	53.77	34.67	89.44	0.13
		16	0.99	53.74	21.35	76.08	0.14
		32	0.97	54.18	17.72	**72.87**	0.15
	64	**4**	0.98	28.11	42.71	71.80	0.13
		8	0.97	26.87	39.42	67.26	0.14
		16	1.02	28.10	24.79	53.91	0.15
		32	1.01	28.68	20.48	50.17	0.17
		64	1.02	31.25	15.54	**47.81**	0.20
	128	**4**	1.29	14.34	51.60	67.23	0.14
		8	0.97	14.28	45.30	60.55	0.15
		16	0.93	14.68	29.80	45.41	0.16
		32	0.99	16.22	26.83	44.04	0.20
		64	1.00	16.79	20.16	37.95	0.23
		128	0.98	19.12	15.28	**35.38**	0.29
8192	**8**	**4**	1.22	208.31	14.40	223.93	0.23
		8	1.23	205.89	13.34	**220.46**	0.23
	16	**4**	1.21	118.44	29.73	149.38	0.27
		8	1.24	104.65	28.54	134.43	0.25
		16	1.22	106.15	16.57	**123.94**	0.26
	32	**4**	0.99	51.76	41.03	93.78	0.24
		8	0.98	52.07	37.05	90.10	0.26
		16	1.01	53.10	21.98	76.09	0.27
		32	0.95	54.13	17.20	**72.28**	0.30
	64	**4**	0.99	27.21	46.34	74.54	0.26
		8	1.01	27.91	43.22	72.14	0.28
		16	1.04	28.85	25.47	55.36	0.30
		32	1.05	29.68	20.74	51.47	0.34
		64	0.91	31.02	15.94	**47.87**	0.40
	128	**4**	0.94	13.96	53.98	68.88	0.27
		8	0.96	14.20	46.75	61.91	0.29
		16	0.95	14.70	29.45	45.10	0.31
		32	1.00	17.01	25.32	43.33	0.40
		64	0.95	17.12	19.80	37.87	0.46
		128	0.94	18.57	15.30	**34.81**	0.58

Tabela 5: 4GB - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	**Mapper Nodes (# Map)**	**Reducer Nodes (# Red)**	**Input (*I*): 4 GB**				
			Tempo do Coordinator (s)	**Tempo dos Mappers (s)**	**Tempo dos Reducers (s)**	**Tempo Total (s)**	**Custo (USD)**
1024	**128**	**4**	1.23	49.37	134.92	185.52	0.11
		8	1.23	49.62	114.67	165.52	0.12
		16	1.32	49.98	78.90	130.20	0.13
		32	1.31	51.03	61.47	113.81	0.14
		64	1.26	52.38	47.37	101.01	0.16
		128	1.20	54.11	32.58	**87.89**	0.18
2048	**64**	**4**	1.18	53.07	77.37	131.62	0.12
		8	1.27	53.03	68.35	122.65	0.13
		16	1.21	57.12	40.47	98.80	0.14
		32	1.22	55.43	32.11	88.76	0.15
		64	1.24	57.64	25.06	**83.94**	0.18
	128	**4**	1.19	28.07	87.25	116.51	0.13
		8	1.23	29.01	83.02	113.26	0.15
		16	1.25	29.28	48.34	78.87	0.15
		32	1.22	30.76	40.63	72.61	0.17
		64	1.24	31.35	30.16	62.75	0.20
		128	1.21	33.83	22.85	**57.89**	0.24
4096	**32**	**4**	1.21	102.64	53.88	157.73	0.23
		8	1.20	104.18	52.39	157.77	0.25
		16	1.23	102.79	31.15	135.17	0.25
		32	1.24	106.10	26.69	**134.03**	0.28
	64	**4**	1.44	53.07	72.44	126.95	0.25
		8	1.24	53.40	71.14	125.78	0.27
		16	1.21	53.65	43.15	98.01	0.28
		32	1.25	55.86	35.30	92.41	0.31
		64	1.23	57.02	26.73	**84.98**	0.36
	128	**4**	1.21	27.73	88.05	116.99	0.26
		8	1.24	28.18	76.75	106.17	0.28
		16	1.22	29.28	47.88	78.38	0.30
		32	1.22	29.40	40.00	70.62	0.34
		64	1.20	36.08	31.53	**68.81**	0.44
		128	1.16	43.90	26.15	71.21	0.60
8192	**16**	**4**	1.20	188.58	7.92	197.70	0.41
		8	1.28	190.38	6.04	197.70	0.41
		16	1.37	190.95	4.30	**196.62**	0.42
	32	**4**	1.25	95.34	19.46	116.05	0.42
		8	1.25	95.66	13.14	110.05	0.42
		16	1.38	95.04	8.02	104.44	0.42
		32	1.21	96.83	5.33	**103.37**	0.44
	64	**4**	1.23	47.22	31.61	80.06	0.42
		8	1.21	47.76	19.74	68.71	0.43
		16	1.35	47.88	12.73	61.96	0.44
		32	1.23	48.54	9.02	58.79	0.45
		64	1.22	50.75	6.35	**58.32**	0.49
	128	**4**	1.26	23.08	37.05	61.39	0.41
		8	1.24	23.65	26.48	51.37	0.43
		16	1.36	24.15	18.14	43.65	0.45
		32	1.34	24.73	12.85	38.92	0.48
		64	1.71	26.62	10.21	**38.54**	0.54
		128	1.36	29.12	8.69	39.17	0.65

{2048, 4096, 8192} alcançaram tempos similares entre si, por exemplo, para # **Map** = 64 e # **Mem** = {2048, 4096, 8192}.

Vale ressaltar que # **Map** influencia o tempo de execução da fase *REDUCER* para qualquer # **Mem**. Quanto mais *mappers*, mais fragmentos são gerados, sendo estes posteriormente recuperados e processados pelos *reducers*. Por outro lado, quanto menos *mappers*, maior o risco de estouro de memória na etapa *Map* do *MARLA*. Dentre as diferentes configurações de # **Mem**, os melhores tempos de execução da aplicação foram obtidos com o maior # **Map** e o maior # **Red** para todos os cenários especificados. Dentre todos os cenários especificados para as execuções com a entrada de 2GB, o que se saiu melhor em termos de tempo de execução, foi a especificação de # **Mem** = 2048, com 128 *mappers* e 128 *reducers* executando em 33,85 segundos.

5.2.2 Execuções do *Word-Count-MARLA* com entrada de 4GB de dados

A **Tabela 5** apresenta as médias dos tempos de execução e respectivos custos monetários para a entrada de 4 *GB*. Assim como para a entrada de 2GB, foram consideradas cinco execuções por cenário. Foram também coletados os tempos associados aos agentes de cada fase no *MARLA*.

As conclusões das análises realizadas para as execuções com o arquivo de entrada de 4GB, possui algumas similaridades quanto as realizadas com o arquivo de 2GB. É possível verificar algumas comparações entre essas execuções:

1. Para o arquivo de 4GB, repetiu-se o mesmo comportamento em relação aos tempos de execução com # **Mem** = 1024 MB, que por sua vez foram em torno de 73% maiores que nas demais configurações de memória.

2. Os tempos de execução da fase *MAPPER* para o arquivo de 4GB aumentam de forma proporcional ao aumento do tamanho do arquivo.

3. Os melhores tempos de execução da aplicação foram obtidos com o maior # **Map** e o maior # **Red**, para praticamente todos os cenários considerados.

Por outro lado, os tempos de execução da fase *REDUCER* não mantiveram o aumento em proporção ao tamanho do arquivo.

Apesar de a maioria dos melhores tempos de execução serem com a maior quantidade de *reducers* e de *mappers*, a melhor combinação utilizando o arquivo de 4GB de entrada

foi com # **Mem** = 8192 MB, # **Map** = 128 e # **Red** = 64, atingindo o tempo total de execução de 38,54 segundos.

Com base nas comparações, as conclusões comparativas podem ser melhor visualizadas na **Tabela 6**, que apresenta um sumário dos tempos médios de execução para as entradas de 2GB e 4GB, utilizado-se o cenário em que # **Mem** = 2048 e # **Map** = {64, 128}. Além disso, para a comparação mencionada, foram removidos os tempos da fase do *COORDINATOR* (pois são sempre em torno de 1s, em todos os casos) e os custos, a fim de destacar os tempos da fase *MAPPER* e *REDUCER*.

Tabela 6: Tempos médios de execução *Word-Count-MARLA* 2GB e 4GB - comparação.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Tempo Mapper (s)		Tempo Reducer (s)		Tempo Total (s)	
			(*I*): 2 GB	(*I*): 4GB	(*I*): 2 GB	(*I*): 4GB	(*I*): 2 GB	(*I*): 4GB
2048	**64**	**4**	28.02	53.07	42.60	77.37	71.60	131.62
		8	28.33	53.03	39.57	68.35	68.92	122.62
		16	28.71	57.12	24.91	40.47	54.74	98.80
		32	29.40	55.43	20.27	32.11	50.73	88.76
		64	31.04	57.64	15.67	25.06	**47.64**	83.94
	128	**4**	14.51	28.07	51.90	87.25	67.40	116.51
		8	14.62	29.01	44.87	83.02	60.44	113.26
		16	14.84	29.28	29.66	48.34	45.44	78.87
		32	15.24	30.76	24.51	40.63	40.74	72.61
		64	16.22	31.35	20.82	30.16	37.99	62.75
		128	17.51	33.83	15.31	22.85	**33.85**	57.89

É perceptível que o tempo de execução da fase *MAPPER* aumenta à medida em que # **Red** aumenta. Isso ocorre porque cada *mapper* gera um fragmento de dados distinto para que cada *reducer* possa processá-lo. Consequentemente, à medida que o número de *reducers* aumenta, mais fragmentos precisam ser gerados e enviados para o armazenamento no *bucket* S3.

Ainda analisando as comparações apresentadas na **Tabela 6** em relação ao tempo de execução de cada fase, é evidente que o aumento no tempo das execuções **não** é proporcional em todas as fases. Na fase *MAPPER*, pode-se dizer que o tempo de execução aumenta proporcionalmente ao tamanho do arquivo, porém na fase *REDUCER*, o tempo de execução fica em média apenas 65% maior, quando comparadas as execuções do arquivo de entrada de 4 GB e de 2 GB. Esses aumentos podem ser descritos da seguinte maneira:

1. Fase de Mapeamento (*MAPPER*): Em termos percentuais, a média de aumento é de 93%, com o maior aumento atingindo 102% e o menor em 86%, corroborando para a afirmação de que o aumento é proporcional ao aumento do tamanho do arquivo de entrada.

2. Fase de Redução (*REDUCER*): Em termos percentuais, a média de aumento é de 65%, com o maior aumento chegando a 85% e o menor ficando em 45%.

3. Tempo total das duas fases: Em termos percentuais, a média de aumento é de 77%, com o maior aumento alcançando 87% e o menor em 65%.

Tabela 7: 10GB - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 10 GB Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
2048	**256**	**4**	1.04	47.39	94.80	143.23	0.42
		8	1.06	45.61	81.39	128.06	0.41
		16	1.31	48.80	55.41	105.52	0.45
		32	1.01	48.68	40.36	90.05	0.46
		64	1.04	47.39	38.03	86.46	0.49
		128	0.99	51.86	30.74	83.59	0.57
	512	**4**	1.02	24.35	144.81	170.18	0.43
		8	1.03	24.39	114.32	139.74	0.45
		16	1.07	25.26	87.50	113.83	0.48
		32	0.97	21.91	68.11	90.99	0.45
		64	1.02	36.33	65.01	102.36	0.76
		128	1.30	36.15	54.78	92.23	0.85
4096	**128**	**4**	1.24	70.63	64.32	136.19	0.62
		8	1.28	70.71	55.57	127.56	0.63
		16	1.39	71.38	33.75	106.52	0.65
		32	1.21	71.75	26.78	99.74	0.67
		64	1.23	73.26	23.51	98.00	0.73
		128	1.38	75.81	18.73	95.92	0.81
	256	**4**	1.05	34.52	91.89	127.46	0.61
		8	1.03	34.62	78.02	113.67	0.63
		16	1.05	35.13	57.72	93.90	0.66
		32	0.98	36.54	41.29	78.81	0.71
		64	0.98	43.36	37.69	82.03	0.90
		128	0.99	40.35	30.52	71.86	0.95
	512	**4**	1.04	18.60	140.55	160.19	0.67
		8	1.03	18.38	116.01	135.42	0.69
		16	1.03	18.20	88.59	107.82	0.72
		32	1.01	20.48	67.06	88.55	0.84
		64	1.01	24.15	63.00	88.16	1.09
		128	0.95	33.24	53.03	87.22	1.59

5.2.3 Execuções com entrada 10GB

Diante dos resultados, foram elaborados mais alguns cenários em busca do entendimento dessas análises. Portanto, foram executados testes e coletados as médias dos tempos de execução e respectivos custos monetários para um arquivo de entrada de 10 GB. Os cenários seguiram as seguintes definições: # **Mem** = 2048 e 4096, # **Map** = 128, 256, 512 e # **Red** = 4 até 128 em potências de 2. A quantidade de memória foi escolhida assim, pois o foco estava em avaliar o tempo de execução da fase *REDUCER*, descartando # **Mem** = 1024 e 8192. Note que o número de *mappers* foi aumentado em relação aos cenários de 2 GB e 4 GB a fim de destacar o aumento do tempo de execução na etapa *REDUCER* ao variar # **Map**. Aumentou-se também o tamanho do arquivo, para verificar o aumento do tempo de execução da fase *REDUCER*, que nas análises anteriores, não se mostrou proporcional (média de 65% de aumento para um arquivo 100% maior).

A **Tabela 7** apresenta as médias dos tempos de execução e respectivos custos monetários para a entrada de 10 GB. Nesta análise experimental também foram consideradas cinco execuções por cenário. Foram também coletados os tempos associados aos agentes de cada fase no *MARLA*.

De acordo com os resultados coletadas das execuções disponibilizados na **Tabela 7**, o aumento no tempo da fase *REDUCER* de 60,62% em média, com o aumento do # **Map**, apesar da diminuição de tempo da fase *MAPPER* de aproximadamente de 58%. Este aumento é tão significativo, que o aumento no tempo total de execução pode chegar até 25%. Desta forma é possível entender que aumentar a # **Map** nem sempre irá melhorar o tempo de execução.

Tabela 8: Tempos médios de execução *Word-Count-MARLA* 4GB e 10GB - comparação.

Lambda Function Memory (MB) (# Mem)	**Mapper Nodes (# Map)**	**Reducer Nodes (# Red)**	**Tempo Mapper (s)**		**Tempo Reducer (s)**		**Tempo Total (s)**	
			(*I*): 4 GB	**(*I*): 10 GB**	**(*I*): 4 GB**	**(*I*): 10 GB**	**(*I*): 4 GB**	**(*I*): 10 GB**
4096	**128**	**4**	28.07	70.63	87.25	64.32	116.51	136.19
		8	29.01	70.71	83.02	55.57	113.26	127.56
		16	29.28	71.38	48.34	33.75	78.87	106.52
		32	30.76	71.75	40.63	26.78	72.61	99.74
		64	31.35	73.26	30.16	23.51	62.75	98.00
		128	33.83	75.81	22.85	18.73	**57.89**	**95.92**

A **Tabela 8** apresenta a comparação dos tempos médios de execução para as entradas de 4GB e 10GB, utilizado-se o cenário em que # **Mem** = 4096 e # **Map** = 128. Além disso, para a comparação mencionada, foram removidos os tempos da fase do *COORDINATOR* e os custos monetários, a fim de destacar os tempos da fase *MAPPER* e *REDUCER*.

Ao analisar os tempos da fase de mapeamento (MAPPER), fica evidente que eles seguem a proporção do tamanho do arquivo de entrada – tempos de execução para entrada de 10 GB é cerca de 2,5 vezes maior que os tempos para entrada de 4 GB. Porém para a fase *REDUCER*, nota-se um comportamento bem diferente do que se havia constatado nos resultados das execuções dos arquivos de 2 GB e 4 GB, no qual todos os tempos de execução da fase *REDUCER* para o arquivo de 10 GB foram menores que o de 4 GB. Esse comportamento foi identificado como uma diferença no conteúdo do arquivo, pois o arquivo de 4 GB, foi formado através da duplicação e concatenação do arquivo de 2GB, utilizado nos testes, com ele mesmo. No entanto, o arquivo de 10 GB foi formado por um conteúdo totalmente diferente. Portanto, para entendermos esse comportamento, realizamos alguns testes a fim de expor como a diferença de conteúdo poderia impactar a etapa *REDUCER*.

5.2.4 Impacto do Conteúdo da Entrada

Os experimentos na Seção 5.2 mostraram que os tempos de execução da fase *REDUCER*, para a entrada com o arquivo de 10GB em todos os cenários, foram menores que os tempos de execução obtidos nos experimentos com o arquivo de entrada de 4GB. Além disso, foi mencionada a diferença do conteúdo entre os dois arquivos.

Com o objetivo de compreender melhor como o conteúdo do arquivo pode afetar os tempos das fases do MARLA considerando a aplicação *WordCount* , foram conduzidos testes utilizando o mesmo tamanho de arquivo de entrada, mas variando-se a frequência das palavras presentes.

Os testes foram realizados com diferentes cenários, incluindo um arquivo de 2GB base, chamado aqui de **ArqBase**, formado por textos de livros disponibilizados pelo Projeto Gutenberg (PROJECT. . . , 2023), que difere daquele utilizado na Seção 5.2 anterior. Além disso, foram criadas quatro variações desse arquivo base. Em cada uma delas, uma palavra foi escolhida para ser repetidamente inserida até que se atingisse um tamanho correspondente a 25%, 50%, 75% do conteúdo total do arquivo base, enquanto o restante, foi preenchido com texto proveniente do arquivo base. Estes arquivos foram denominados **Arq25**, **Arq50** e **Arq75**, respectivamente em relação aos percentuais.

As configurações escolhidas para os testes foram as mesmas utilizadas na Seção 5.2, exceto pela quantidade de memória alocada, que foi fixada em # **Mem** = 2048MB, pois a análise foi feita a fim de entender o impacto do conteúdo de entrada, dessa forma analisamos os tempos de diferentes tamanhos de entrada, com a mesma quantidade de memória alocada para a função Lambda.

As médias dos tempos de execução e respectivos custos monetários para os arquivos de entrada **ArqBase**, **Arq25**, **Arq50** e **Arq75** são apresentados nas Tabela 9, 10, 11 e 12, respectivamente.

Após analisar os resultados das execuções, é possível notar que na medida em que aumentamos a porcentagem de palavras repetidas, o tempo da fase *REDUCER* vai diminuindo, em aproximadamente 79%, 58% e 39% do tempo alcançado na execução do arquivo base (**ArqBase**), para os arquivos **Arq25**, **Arq50** e **Arq75** respectivamente. Já os tempos de execução da fase *MAPPER* se mantém similares aos do arquivo base.

Além disso, quando comparamos os resultados das execuções tendo como entrada

Tabela 9: ArqBase - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 2 GB - ArqBase Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
2048	32	4	0.94	53.71	14.43	69.08	0.06
		8	0.94	54.55	12.04	67.54	0.06
		16	0.99	55.14	8.00	64.13	0.06
		32	0.98	54.45	5.81	61.25	0.06
	64	4	0.98	27.15	19.43	47.56	0.06
		8	0.99	27.27	16.93	45.19	0.06
		16	0.97	27.86	12.02	40.85	0.07
		32	0.95	29.16	9.98	40.09	0.07
		64	1.00	30.29	8.71	40.01	0.08
	128	4	0.96	13.54	29.58	44.09	0.06
		8	0.96	14.06	25.38	40.40	0.07
		16	0.89	19.56	19.03	39.49	0.09
		32	0.96	15.37	15.61	31.94	0.08
		64	0.96	16.60	13.49	31.05	0.10
		128	0.93	21.56	12.72	35.21	0.15

Tabela 10: Arq25 - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 2 GB - Arq25 Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
2048	32	4	1.10	56.42	11.92	69.44	0.06
		8	1.12	56.97	9.5	67.59	0.06
		16	1.20	58.38	6.65	66.23	0.07
		32	1.07	59.53	5.07	65.67	0.07
	64	4	0.96	27.58	15.59	44.13	0.06
		8	1.00	28	13.7	42.70	0.06
		16	0.94	32.64	9.79	43.37	0.07
		32	0.98	31.79	7.96	40.73	0.08
		64	0.98	31.11	7.49	39.58	0.08
	128	4	0.94	14.46	23.88	39.28	0.06
		8	0.96	14.25	20.71	35.92	0.07
		16	0.99	15.3	16.02	32.31	0.07
		32	0.96	16.51	13.24	30.71	0.08
		64	0.93	17.42	13	31.35	0.10
		128	0.99	20.76	13.05	34.80	0.14

Tabela 11: Arq50 - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 2 GB - Arq50 Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
2048	32	4	1.24	64.19	7.85	73.28	0.07
		8	1.21	64.87	6.41	72.49	0.07
		16	1.34	59.58	4.96	65.88	0.07
		32	1.24	57.99	4.03	63.26	0.07
	64	4	1.01	27.92	11.69	40.62	0.06
		8	0.98	28.22	10.39	39.59	0.06
		16	1.03	28.55	7.81	37.39	0.07
		32	0.99	29.71	6.89	37.59	0.07
		64	1.03	31.21	6.18	38.42	0.08
	128	4	1.02	14.37	18.58	33.97	0.06
		8	1.01	15.32	16.28	32.61	0.07
		16	0.99	15.16	14.05	30.20	0.07
		32	1.01	17.05	11.32	29.38	0.08
		64	1.00	17.54	11.64	30.18	0.10
		128	1.00	19.67	10.26	30.93	0.13

Tabela 12: Arq75 - Tempos e custos médios de execução do *Word-Count-MARLA*.

Lambda Function Memory (MB) (# Mem)	Mapper Nodes (# Map)	Reducer Nodes (# Red)	Input (*I*): 2 GB - Arq75 Tempo do Coordinator (s)	Tempo dos Mappers (s)	Tempo dos Reducers (s)	Tempo Total (s)	Custo (USD)
2048	32	4	1.19	54.62	5.12	60.93	0.06
		8	1.22	54.86	4.48	60.56	0.06
		16	1.22	55.36	4.81	61.39	0.06
		32	1.23	57.01	4.84	63.08	0.07
	64	4	1.04	27.51	7.59	36.14	0.06
		8	0.99	27.67	6.70	35.36	0.06
		16	1.00	27.83	6.09	34.92	0.06
		32	1.01	28.87	5.13	35.01	0.07
		64	0.98	30.45	5.38	36.81	0.08
	128	4	0.99	13.83	13.65	28.47	0.06
		8	0.95	13.94	11.91	26.80	0.06
		16	1.00	15.22	10.10	26.32	0.07
		32	0.95	16.40	9.97	27.32	0.08
		64	1.02	17.22	9.05	27.29	0.09
		128	0.95	20.28	10.33	31.56	0.13

ArqBase (de 2GB) com aquelas tendo como entrada o arquivo de 2GB da Seção 5.2, para os cenários em que a # **Mem** = 2048MB, é nítida a diferença dos tempos de execução da etapa *REDUCER*, uma vez que os conteúdos são diferentes, sendo **ArqBase** formado por textos distintos.

Essa diferença de tempo pode ser atribuída à forma em que os fragmentos de dados são distribuídos para os *reducers*. Na Subseção 4.1.2, é mencionado que durante o processo de mapeamento dos fragmentos, o agrupamento destes é baseado no primeiro caractere da chave de cada par de chave/valor. Isso ocorre porque cada *reducer* é encarregado de processar os pares de chave/valor cujo primeiro caractere da chave pertence a um intervalo definido de caracteres imprimíveis da tabela ASCII. Esses intervalos são definidos com base no número de *reducers*, tal que, com dois *reducers* por exemplo, cada um seria responsável pelo intervalo equivalente a metade dos caracteres imprimíveis da tabela ASCII.

Como resultado, quando os conteúdos dos arquivos de entrada são distintos, diferentes distribuições de chaves são realizadas, levando a diferentes cargas de processamento para cada *reducer*, influenciando no tempo de execução da fase *REDUCER*.

5.3 Análise dos Resultados com Spark

Para a análise do *Word-Count-Spark*, foi selecionada a instância *r5.large* (com 2 *vCPUs* e 16 *GB* de Memória) para a formação de diferentes tamanhos de cluster *Spark*, pois, segundo a *Amazon* (AMAZON, 2023b), é um tipo de MV ideal para *caches* distribuídos na memória e análises de *Big Data*. Ademais, em (NUNES et al., 2021) foram observadas vantagens na utilização de MVs otimizadas para memória para este tipo de aplicação.

Considerando que o tempo de execução dos experimentos foram relativamente baixos (menores que 3 minutos no pior dos casos), o mercado *Spot* foi utilizado em todos os experimentos, sem que houvesse interrupção da execução, minimizando também custos de utilização. Além disso o custo monetário para cada instância foi fixado em 0,048 *USD* por hora de uso. Foram desconsiderados os custos associados ao tempo de construção de cada cluster na *AWS EC2* e ao armazenamento e recuperação de dados no *Amazon S3*. Cada cluster *Spark* foi formado a partir de um nó *Master* (*Driver Program*) conectado a um ou mais nós *Workers* (cujo número, é denotado por # **W**). Para cada *Worker*, foi definido apenas um *Spark Executor*. Para cada configuração de *cluster Spark*, a quantidade de *Map Tasks* (# **Map**) foi definida conforme a quantidade total de *vCPUs* disponíveis no respectivo *cluster*, até um máximo de 128 *vCPUs*, enquanto que a quantidade de *Reduce Tasks* (# **Red**) foi variada considerando todos os possíveis divisores de # **Map**, *i.e.*, # **Red** $= \{r \in \mathbb{N}^* | r \backslash \#\ \mathbf{Map}\}$. Para cada cenário formado, os volumes de dados de entrada (I) considerados foram 2 *GB* e 4 *GB*, pois são estes que possuem a maior quantidade de cenários executados durante o experimento do MARLA.

A Tabela 13 exibe os tempos médios e custos resultantes de cinco execuções por cenário. Ela destaca as combinações de # **W**, # **Map** e # **Red** que resultaram nos menores tempos de execução da aplicação.

Note que a configuração $\langle \#\ \mathbf{W} = 64, \#\ \mathbf{Map} = 128, \#\ \mathbf{Red} = 32 \rangle$ obteve o melhor desempenho de tempo para um input de 2 *GB*, com um custo de 0,05 *USD* e tempo de execução de 64,20 segundos. Já para uma entrada de 4 *GB*, a mesma configuração $\langle \#\ \mathbf{W} = 64, \#\ \mathbf{Map} = 128, \#\ \mathbf{Red} = 32 \rangle$ alcançou o melhor tempo de execução, com um custo de 0,06 *USD* e um tempo de execução de 74,24 segundos.

Tabela 13: Tempos e custos médios de execução do *Word-Count-Spark*.

Quantidade de Workers (# W)	**Input (I): 2 GB**				**Input (I): 4 GB**			
	# Map	**# Red**	**Tempo (s)**	**Custo (USD)**	**# Map**	**# Red**	**Tempo (s)**	**Custo (USD)**
1	2	2	362,70	0,01	2	2	693,45	0,02
2	4	4	274,70	0,01	4	4	364,03	0,01
3	6	6	145,68	0,01	6	6	253,98	0,01
4	8	8	119,55	0,01	8	8	201,89	0,01
5	10	10	107,09	0,01	10	10	170,75	0,01
6	12	12	95,07	0,01	12	12	150,72	0,01
7	14	14	87,73	0,01	14	14	141,47	0,02
8	16	16	85,20	0,01	16	16	130,01	0,02
9	18	18	79,13	0,01	18	18	117,98	0,02
10	20	20	81,18	0,01	20	20	111,64	0,02
11	22	22	75,54	0,01	22	22	111,24	0,02
12	24	24	76,21	0,01	24	24	102,89	0,02
13	26	26	71,29	0,01	26	26	104,27	0,02
14	28	28	71,58	0,01	28	28	101,93	0,02
15	30	30	74,28	0,02	30	30	98,08	0,02
16	32	32	75,93	0,02	32	32	97,27	0,02
32	64	32	73,77	0,03	64	64	98,05	0,04
64	128	32	**64,20**	0,05	128	32	**74,24**	0,06

5.4 Desempenho e Custo: *Word-Count* MARLA e Spark

Atualmente, a capacidade máxima de memória disponível por função Lambda na *AWS Lambda* é de 10.240 *MB* (10 *GB*). Esta restrição motivou o uso da instância *r5.large* para a construção do cluster *Spark* na *AWS EC2*, visto que é a instância *memory optimized* com capacidade de memória (16 *GB*) mais próxima daquela. Quanto à capacidade de processamento, não foi possível determinar configurações que fossem totalmente equivalentes. Na *AWS EC2*, o cliente é capaz de obter a informação da quantidade de *vCPUs* disponíveis por instância de um modo transparente (em particular, 2 *vCPUs* para a *r5.large*). Conhecimento similar já não é possível no *AWS Lambda*, pois esta informação não se encontra disponibilizada oficialmente. Sabe-se apenas que, dependendo da quantidade de memória alocada, são oferecidas de 1 a 6 *vCPUs* por função Lambda (AMAZON, 2020, 2023d).

Para a análise comparativa de custos computacionais utilizando *FaaS* (*MARLA*) e *IaaS* (*Spark*), selecionamos os melhores tempos de execução da aplicação *Word-Count*, considerando os volume de dados de entrada de 2 *GB* e 4 *GB*. A **Tabela 14** sumariza tais resultados. Com o *MARLA* na *AWS Lambda* foi possível executar a aplicação mais rapidamente: aproximadamente 47,3% para a entrada de 2 *GB* e 49% para a entrada de 4 *GB*. Em contrapartida, com o *Spark* na *AWS EC2* foi possível executar a aplicação de modo mais econômico: aproximadamente 64,3% para a entrada de 2 *GB* e 88,8% para a entrada de 4 *GB*. É importante mencionar que foram utilizadas instâncias *Spot* para a execução com o *Spark*. Se fossem utilizadas instâncias *On Demand*, a diferença de custo teria sido menor.

Tabela 14: MARLA Vs. Spark: melhores tempos médios de execução.

Ambiente	Quantidade de Mappers	Quantidade de Reducers	Lambda Function Memory (MB)	Input (GB)	Tempo (s)	Custo (USD)
MARLA + AWS Lambda	128	128	2048	2	33,85	0,14
		64	8192	4	38,54	0,54
Spark + AWS EC2	128	32	N/A	2	64,2	0,05
				4	74,24	0,06

Alguns fatores não foram considerados nesta análise, tais como: o tempo entre a chamada da função lambda e sua efetiva execução (*cold start*), tratamento de recursos ociosos na *AWS EC2*, e utilização mais eficiente do *Amazon S3*. Do ponto de vista do cluster *Spark*, deve-se evitar ao máximo a subutilização de recursos, por exemplo, por falta de trabalho a ser executado. É preciso avaliar as vantagens e desvantagens em manter um *cluster* configurado, ainda que os clientes sejam cobrados apenas pelos volumes de

armazenamento *EBS* e de endereço *IP* público reservado que estejam associados a cada instância interrompida (neste caso as instâncias de máquinas virtuais do cluster ficariam em estado *Stopped*). Do ponto de vista do *MARLA*, deve-se evitar ao máximo o estouro do tempo de execução permitido para cada função Lambda, atualmente limitado a quinze minutos. Além disso, é preciso analisar as vantagens e desvantagens de realizar leituras e escritas de dados parciais no *Amazon S3* durante o processamento do *MapReduce* com *MARLA*, pois estas ações também possuem custos monetários associados.

6 Conclusões e Trabalhos Futuros

Este estudo oferece uma análise comparativa entre as execuções de aplicação *MapReduce* usando os *frameworks MARLA* e *Spark* na AWS, analisando os tempos de execução e os custos associados ao uso de cada um dos frameworks mencionados, além de explorar as limitações atuais do AWS Lambda e suas possíveis configurações. A análise realizada neste trabalho indica que adotar o modelo *FaaS* para execução de aplicação *MapReduce* através do MARLA pode ser vantajosa.

No MARLA, o usuário se concentra apenas na lógica das funções *Map* e *Reduce*, enquanto o *framework* se responsabiliza pela criação da infraestrutura necessária para a execução da aplicação. Em contrapartida, na abordagem com o *Spark* no modelo *IaaS*, além de especificar as funções *Map* e *Reduce*, é essencial realizar manualmente a implantação do *cluster Spark*, incluindo a configuração de *Workers* e *Executors*, conforme detalhado na Subseção referente à implementação do *Spark*.

No que diz respeito à codificação da aplicação, ambas as abordagens apresentam desafios semelhantes. É necessário escolher a estrutura de dados adequada e aplicar as operações corretas para gerar e processar os pares de chave-valor usados no modelo *MapReduce*. A avaliação experimental da aplicação *Word-Count* destacou benefícios no desempenho de execução para a versão *MARLA* em comparação com a versão *Spark*. No entanto, vale ressaltar que essa melhoria de desempenho é acompanhada por um custo monetário maior. Em termos de desempenho, o *MARLA* (*FaaS*) executou mais rápido a aplicação em aproximadamente 47,3% para a entrada de 2 GB e 49% para a entrada de 4 GB, enquanto que para o *Spark* (*IaaS*), o custo monetário das execuções ficou mais econômico em aproximadamente 64,3% para a entrada de 2 GB e 88,8%, quando comparado com o *MARLA*.

Apesar do *FaaS* propor um desacoplamento do usuário em relação à infraestrutura e à execução, ainda é necessário especificar algumas características do modelo *MapReduce*, tais como o número de *mappers* e *reducers* que afetam custo monetário e desempenho.

Além disso, a definição da quantidade de memória alocada por função Lambda é um outro fator que também afeta diretamente o custo monetário e o desempenho.

Em relação ao tamanho de memória, o usuário inexperiente pode especificar uma quantidade que leva ao estouro de memória. Para tal, uma análise exploratória foi realizada no intuito de derivar a quantidade mínima de *mappers* que leve a execução com sucesso da aplicação *WordCount*. A análise consistiu em descobrir, para uma determinada quantidade de memória alocada, o menor número de *mappers* necessário para executar a aplicação com sucesso, e dessa forma inferir uma proporção entre a quantidade de memória alocada para a função Lambda e a quantidade de dados que um único *mapper* é capaz de processar. Portanto, a quantidade de dados que um *mapper* é capaz de processar, variou de 3,45% até 3,70%, utilizando o *MARLA* nos cenários especificados na seção anterior.

Dessa forma, ao executar aplicação *MapReduce* com *FaaS*, no geral é interessante que o usuário tenha algum conhecimento sobre a aplicação e *framework* para que se tenha uma boa configuração e consequente custo e desempenho satisfatório. Por exemplo, com base nos experimentos e análises realizados nesse trabalho, utilizando o *MARLA* especificamente, foi possível notar que nem sempre ao aumentar o número de *mappers* havia ganho no desempenho. Da mesma forma acontece para o aumento de memória alocada na função Lambda, que a partir de uma determinada quantidade de memória alocada, o desempenho se mantinha, e em contrapartida o custo aumentava. Ademais, a análise que verifica o impacto de diferentes conteúdos de entrada, com o mesmo tamanho (subseção 5.2.4), revelou que o tempo da fase *REDUCER* fica nitidamente diferente, e pode influenciar bastante no tempo total de execução da aplicação.

Como trabalho futuro, pretende-se elaborar um modelo de otimização da configuração do ambiente *MARLA* na *AWS Lambda* dada uma aplicação, visando a redução de tempo e custo monetário associados à sua execução. Além disso, é também importante avaliar *benchmarks* de *Big Data*, variando os dados de entrada e realizar comparações de desempenho do *MARLA* com *frameworks* similares para *FaaS*.

REFERÊNCIAS

AMAZON. **Amazon AWS - EC2 - Pricing**. Disponível em: <https://aws.amazon.com/pt/ec2/pricing/>. Acesso em: 15 set. 2023.

_____. **Amazon AWS - Services - Instance types**. Disponível em: <https://aws.amazon.com/pt/ec2/instance-types/>. Acesso em: 15 set. 2023.

_____. **Amazon AWS - Services - Lambda help**. Disponível em: <https://aws.amazon.com/pt/lambda/faqs/>. Acesso em: 15 ago. 2023.

_____. **AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions**. [S. l.: s. n.], 2020. Disponível em: <https://aws.amazon.com/pt/about-aws/whats-new/2020/12/aws-lambda-supports-10gb-memory-6-vcpu-cores-lambda-functions/>. Acesso em: 1 de dez. de 2023.

_____. **Cotas Lambda**. [S. l.: s. n.], 2023. Disponível em: <https://docs.aws.amazon.com/pt_br/lambda/latest/dg/gettingstarted-limits.html>. Acesso em: 16 de nov. de 2023.

APACHE. **Apache Spark**. 2022. Disponível em: <https://spark.apache.org/>. Acesso em: 20 abr. 2022.

AWAYSHEH, F. M. et al. Big Data Resource Management & Networks: Taxonomy, Survey, and Future Directions. **IEEE Communications Surveys & Tutorials**, v. 23, n. 4, p. 2098–2130, 2021.

AZEROUAL, Otmane; NIKIFOROVA, Anastasija. Apache Spark and MLlib-Based Intrusion Detection System or How the Big Data Technologies Can Secure the Data. **Information**, v. 13, n. 2, 2022. ISSN 2078-2489. DOI: 10.3390/info13020058. Disponível em: <https://www.mdpi.com/2078-2489/13/2/58>.

BALDINI, Ioana et al. **Serverless Computing: Current Trends and Open Problems**. [S. l.: s. n.], 2017. arXiv: 1706.03178 [cs.DC].

DATABRICKS. **Plataforma de análise unificada do Databricks**. [S. l.: s. n.]. Disponível em: <https://databricks.com/product/unified-analytics-platform. Acesso em: 28 jun. 2023.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. In: PROCEEDINGS of the 6th Symposium on Operating Systems Design and Implementation. San Francisco, CA, USA: USENIX Association, dez. 2004. (OSDI' 04), p. 137–149.

EIVY, Adam; WEINMAN, Joe. Be Wary of the Economics of "Serverless"Cloud Computing. **IEEE Cloud Computing**, v. 4, n. 2, p. 6–12, 2017. DOI: 10.1109/MCC.2017.32.

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

GIMÉNEZ-ALVENTOSA, V.; MOLTÓ, Germán; CABALLER, Miguel. A framework and a performance assessment for serverless MapReduce on AWS Lambda. **Future Generation Computer Systems**, v. 97, p. 259–274, 2019. DOI: 10.1016/j.future.2019.02.057.

GROGAN, Jake et al. A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers. In: [s. l.: s. n.], ago. 2020. P. 58–75. ISBN 978-3-030-56440-7. DOI: 10.1007/978-3-030-56441-4_5.

HASSAN, Hassan; BARAKAT, Saman; SARHAN, Qusay. Survey on serverless computing. **Journal of Cloud Computing**, v. 10, jul. 2021. DOI: 10.1186/s13677-021-00253-7.

JARACHANTHAN, Jananie et al. Astrea: Auto-Serverless Analytics Towards Cost-Efficiency and QoS-Awareness. **IEEE Transactions on Parallel and Distributed Systems**, v. 33, n. 12, p. 3833–3849, 2022. DOI: 10.1109/TPDS.2022.3172069.

JONAS, Eric et al. Occupy the cloud: distributed computing for the 99%. **Proceedings of the 2017 Symposium on Cloud Computing**, 2017.

KAPIL, D.; MISHRA, S.; GUPTA, V. A Performance Perspective of Live Migration of Virtual Machine in Cloud Data Center with Future Directions. **International Journal of Wireless and Microwave Technologies**, v. 12, p. 48–56, jan. 2022. DOI: 10.5815/ijwmt.2022.04.04.

KELLY, Daniel; GLAVIN, Frank G.; BARRETT, Enda. Serverless Computing: Behind the Scenes of Major Platforms. **CoRR**, abs/2012.05600, 2020. arXiv: 2012.05600. Disponível em: <https://arxiv.org/abs/2012.05600>.

KIM, Youngbin; LIN, Jimmy. Serverless Data Analytics with Flint. In: IEEE 11th International Conference on Cloud Computing (CLOUD). Los Alamitos, CA, USA: IEEE Computer Society, jul. 2018. P. 451–455. DOI: 10.1109/CLOUD.2018.00063.

MANNER, Johannes. A Structured Literature Review Approach to Define Serverless Computing and Function as a Service. In. DOI: 10.1109/CLOUD60044.2023.00068.

MEDIUM. **1 Cloud Models — What are the key differences?** 2021. Disponível em: <https://medium.com/nerd-for-tech/1-cloud-series-the-iaas-paas-saas-8faa46124722>. Acesso em: 19 set. 2023.

MUNISWAMAIAH, Manoj; AGERWALA, Tilak; TAPPERT, Charles. Big Data in Cloud Computing Review and Opportunities. **International Journal of Computer Science & Information Technology (IJCSIT)**, v. 11, dez. 2019.

NUNES, Alan 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.

PANKAJ SARASWAT, Swapnil Raj. A-Review-Paper-on-Hadoop-Architecture. **International Journal of Innovative Research in Computer Science and Technology (IJIRCST)**, v. 9, n. 6, p. 96–99, 2021. ISSN 2347 - 5552. DOI: 10.55524/ijircst.2021.9.6.22. Disponível em: <https://www.ijircst.org/view_abstract.php?year=&vol=9&primary=QVJULTYzNQ==>.

PAVLO, Andrew et al. A Comparison of approaches to large-scale data analysis. In: p. 165–178. DOI: 10.1145/1559845.1559865.

PROJECT Gutenberg. Disponível em: <http://www.gutenberg.org>. Acesso em: 15 set. 2023.

REUTER, Anja; BACK, Timon; ANDRIKOPOULOS, Vasilios. Cost efficiency under mixed serverless and serverful deployments. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). [S. l.: s. n.], 2020. P. 242–245. DOI: 10.1109/SEAA51224.2020.00049.

SHVACHKO, Konstantin et al. The Hadoop Distributed File System. In: PROCEEDINGS of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST). [S. l.]: IEEE, 2010. P. 1–10. DOI: 10.1109/MSST.2010.5496972.

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.

ZHANG, Wen et al. Kappa: A Programming Framework for Serverless Computing. In: PROCEEDINGS of the 11th ACM Symposium on Cloud Computing. Virtual Event, USA: Association for Computing Machinery, 2020. (SoCC '20), p. 328–343. ISBN 9781450381376. DOI: 10.1145/3419111.3421277. Disponível em: <https://doi.org/10.1145/3419111.3421277>.