O Balanceamento de Réplicas em um Cluster HDFS com base na Confiabilidade dos Racks

Rhauani Weber Aita Fazul[1], Paulo Vinicius Cardoso[2], Patrícia Pitthan Barcelos[2]
[1]*Laboratório de Sistemas de Computação (LSC)*
[2]*Pós-Graduação em Ciência da Computação (PGCC)*
Universidade Federal de Santa Maria (UFSM)
Santa Maria, Brasil
{rwfazul, pcardoso, pitthan}@inf.ufsm.br

***Resumo*—A replicação de dados é um dos principais mecanismos de tolerância a falhas utilizados pelo HDFS. Porém, a forma de posicionamento das réplicas entre os nodos computacionais afeta diretamente o balanceamento e o desempenho do sistema. O HDFS Balancer é uma solução disponibilizada pelo Apache Hadoop que visa equilibrar a distribuição dos dados. Todavia, sua política de operação atual não permite endereçar demandas de disponibilidade e confiabilidade ao redistribuir as réplicas entre os *racks* do *cluster*. Esse trabalho apresenta uma estratégia de balanceamento customizada para o HDFS Balancer baseada em fatores de confiança, que são calculados para cada *rack* a partir da taxa de falhas de seus nodos. Após detalhar a implementação, conduziu-se uma investigação experimental que permitiu validar e demonstrar a efetividade da estratégia desenvolvida.**

***Index Terms*—replicação, balanceamento, confiança dos racks**

I. Introdução

O Apache Hadoop [1] é uma plataforma de propósito geral dedicada ao armazenamento e ao processamento de dados em larga escala. Por oferecer um robusto sistema de gerência de dados e aplicações em arquiteturas de alto desempenho, tais como *clusters* e *grids*, o Hadoop é muito utilizado por empresas e pesquisadores. A simplicidade de configuração e de uso do *framework*, em conjunto com uma estrutura de desenvolvimento *open-source*, fazem do Hadoop uma poderosa ferramenta para análise e gerência de *big data* [2].

O ecossistema do Hadoop é composto por diversos componentes e ferramentas, capazes de fornecer alta escalabilidade e disponibilidade em ambientes distribuídos. Entre seus módulos primários estão o modelo de programação paralelo MapReduce, o gerenciador de recursos de propósito geral YARN e o sistema de arquivos distribuído HDFS [1]. Sendo a camada de armazenamento do Hadoop, o HDFS permite que inúmeras aplicações – coordenadas pelo YARN – manipulem dados simultaneamente de forma rápida e segura.

Para tal, no HDFS são implementados diversos mecanismos que buscam assegurar a confiabilidade e a disponibilidade do sistema mesmo no advento de falhas. Um dos principais mecanismos de tolerância a falhas adotados pelo HDFS é a replicação de dados. A replicação visa proporcionar confiabilidade por meio de redundância, além de estimular possíveis melhorias de desempenho ao permitir que as aplicações explorem uma maior disponibilidade dos dados. Todavia, a forma de posicionamento das réplicas entre os nodos afeta diretamente o balanceamento e a localidade dos dados no *cluster* [3].

O `HDFS Balancer` [4] é uma solução nativa disponibilizada pelo Hadoop dedicada ao balanceamento de réplicas. Sua política de operação atual, entretanto, não permite que métricas e características específicas do *cluster* e dos *racks* que agrupam os nodos sejam levadas em consideração durante a redistribuição dos dados, de forma que demandas de disponibilidade e confiabilidade podem deixar de ser atendidas.

Este trabalho apresenta uma estratégia que customiza a política de balanceamento atual do `HDFS Balancer` com base na confiabilidade dos *racks* do HDFS. Desse modo, *racks* com uma maior confiabilidade são prioritários ao recebimento de um maior volume de dados durante a movimentação das réplicas. Para tal, para cada *rack* do *cluster*, calcula-se um fator de confiança através da taxa de falha de seus nodos. Com isso, após o balanceamento, *racks* com alta proporção de nodos falhos tendem a manter uma menor quantidade de dados em seus dispositivos de armazenamento.

O artigo está organizado em 7 seções. A Seção II apresenta o HDFS, aprofundando-se no mecanismo de replicação de dados. A Seção III é dedicada às causas e aos problemas do desbalanceamento de réplicas. A Seção IV relata os principais trabalhos relacionados. A Seção V descreve a implementação da solução proposta. A Seção VI exibe e discute os resultados obtidos. Por fim, a Seção VII apresenta as considerações finais e direciona os trabalhos futuros.

II. *Hadoop Distributed File System* (HDFS)

Dentro de um *cluster* HDFS segue-se uma arquitetura mestre-escravo formada por um `NameNode` (NN) e múltiplos `DataNodes` (DNs) [1]. O NN é o servidor mestre que gerencia o *namespace* e os metadados do sistema, controlando o acesso e a distribuição dos arquivos. Enquanto isso, os DNs são os *workers* que realizam efetivamente a recuperação e o armazenamento dos dados. Múltiplas instâncias de DNs possibilitam que a distribuição de dados ocorra em diferentes máquinas do *cluster*, sendo estas conectadas por rede.

Desde sua criação, o HDFS foi otimizado para armazenar dados na escala de *petabytes* [4]. Para lidar com esse volume

Trabalho apoiado pelo Programa de Educação Tutorial (PET/MEC).

de dados massivo, o HDFS define uma estrutura de armazenamento própria, onde, quando um arquivo é inserido no sistema, ao invés de salvá-lo em sua forma original, gera-se uma sequência de blocos. Os blocos consistem em segmentos de dados criados de forma automática a partir do particionamento do arquivo inicial. Todos os blocos, com exceção do último que pode ser menor, possuem um tamanho fixo: 128MB por padrão, a partir da segunda versão do Hadoop [2].

A alta escalabilidade do Hadoop torna possível que os *clusters* possuam milhares de nodos trabalhando em conjunto, responsáveis tanto por computação como por armazenamento de dados. Mesmo que alguns componentes de *hardware* falhem, é importante que o sistema mantenha sua consistência, de modo que blocos comprometidos possam ser restaurados. Para tal, meios que garantam a integridade e a disponibilidade dos dados através da detecção e recuperação mediante falhas são imprescindíveis. O mecanismo de tolerância a falhas mais expressivo no HDFS é a replicação de dados.

A. Replicação de Dados

Atuando como um importante mecanismo de tolerância a falhas (TF) e como uma estratégia para suprir altas demandas de disponibilidade e acesso aos dados, o HDFS implementa a replicação de blocos. A replicação baseia-se na criação de cópias dos blocos de dados presentes no sistema (réplicas), de modo a aumentar a confiabilidade e a disponibilidade de dados com base em redundância.

Os blocos replicados são armazenados em diferentes nodos do *cluster*. Assim, caso um nodo falhe, seus blocos podem ser acessados a partir de um ou mais DNs que mantenham suas réplicas. O número de réplicas a ser gerado para cada bloco é definido por arquivo a partir de um parâmetro configurável, o Fator de Replicação (FR). Uma aplicação pode especificar o FR no momento da criação de um arquivo, sendo possível modificá-lo posteriormente [1].

A replicação garante que, com um FR de n, os dados armazenados no HDFS fiquem seguros mesmo na ocorrência de $n-1$ falhas simultâneas. Todavia, a replicação inicial dos blocos no momento da escrita de um arquivo no HDFS não é suficiente para garantir TF. Ao executar sobre *hardware* comum, o HDFS está sujeito a falhas frequentes. Para proteger e manter a integridade dos dados em cenários onde ocorram falhas consecutivas em um curto intervalo de tempo, o NN precisa monitorar e controlar ativamente o número de réplicas disponíveis e não corrompidas de cada bloco e, quando aplicável, realizar a re-replicação dos dados [2].

Para todos os blocos que necessitem de re-replicação, o NN deve agir e reiniciar o processo de replicação em outros DNs, assim mantendo a conformidade com o FR definido. Para tal, o NN seleciona um DN que contenha a cópia do bloco a ser re-replicado e um DN como destino para a nova réplica. Esta escolha é feita de forma arbitrária e, assim como a replicação, é realizada de forma transparente pelo HDFS. O processo de re-replicação, dentre outros motivos, pode ser originado devido a falhas no funcionamento de algum DN [1].

Falhas na comunicação entre o NN e o DN são identificadas pela ausência de mensagens *heartbeat*: um mecanismo de TF que permite detectar falhas operacionais em DNs [2]. Periodicamente, os DNs enviam *heartbeats* ao NN a fim de notificá-lo sobre seu estado de funcionamento. Caso o NN não receba *heartbeats* de um DN dentro de um determinado intervalo de tempo, o NN marca o DN em questão como inativo. Além de não receber novas requisições de entrada e saída (E/S), o FR dos blocos de um DN inativo é decrementado. Se o número de réplicas de um bloco for inferior ao FR especificado, a re-replicação pode ser disparada pelo NN a partir de cópias sobressalentes armazenadas em algum dos DNs ativos. Desse modo, a conformidade com o FR é restaurada, preservando a disponibilidade dos dados caso novas falhas ocorram.

Ao realizar a replicação e a re-replicação, uma quantidade considerável de blocos deve ser distribuída entre os DNs. O posicionamento desses blocos é importante para assegurar a disponibilidade e o desempenho do sistema. Atuando como o nodo mestre, o NN é encarregado das decisões acerca da replicação dos blocos no HDFS. Para aumentar a disponibilidade e garantir uma melhor comunicação entre as réplicas e os clientes, o NN segue uma política de posicionamento como referência para a distribuição dos dados no *cluster*.

B. Política de Posicionamento de Réplicas (PPR)

Instâncias do HDFS, em geral, são compostas por múltiplos nodos, dispostos em diferentes *racks*. Sendo um sistema que preza pela tolerância a falhas, o HDFS deve evitar a perda de dados mesmo que um *rack* inteiro falhe. Ao ser armazenado, um bloco é replicado com base no FR de seu arquivo. Para cada bloco a ser armazenado no HDFS, o NN precisa selecionar os DNs para o recebimento de suas réplicas. Esta escolha deve ser realizada visando manter a disponibilidade dos blocos em caso de falhas e aprimorar o desempenho do sistema em operações sobre os dados armazenados.

Para tal, o NN guia-se por um modelo inteligente para o posicionamento das réplicas, otimizado para aumentar a disponibilidade e o desempenho do HDFS de acordo com a arquitetura do *cluster* [2]. O modelo corrente, conhecido como Política de Posicionamento de Réplicas (PPR) [1], é aplicado da seguinte forma (Apache Hadoop versão 2): (i) a primeira réplica é armazenada no mesmo nodo do cliente, porém se o cliente HDFS estiver executando fora do *cluster*, um DN é escolhido aleatoriamente pelo NN (embora o sistema evite, na medida do possível, escolher nodos que estejam com alto tráfego de comunicação); (ii) as duas réplicas seguintes são armazenadas em diferentes DNs de um mesmo *rack* remoto, sendo este diferente do *rack* da primeira réplica; (iii) caso o FR seja maior que o padrão (três réplicas por bloco), as réplicas seguintes são posicionadas de forma arbitrária, porém mantendo o número de réplicas por *rack* abaixo do valor limite resultante de $((\text{réplicas} - 1) \; / \; \text{racks}) + 2$.

Em relação à sobrecarga no processo de escrita, que envolve o armazenamento das múltiplas réplicas de um mesmo bloco, o HDFS aplica, quando possível, a técnica de *pipeline* de replicação [2]. A partir da definição dos DNs para o rece-

bimento das réplicas, estabelece-se um *pipeline* onde DNs podem, simultaneamente, receber e encaminhar dados. Além de aprimorar a operação de escrita, o *pipeline* permite que todo processo de replicação seja transparente ao usuário, que apenas precisa interagir com um único nodo.

A replicação também possibilita otimizar operações voltadas à leitura e à recuperação dos dados, onde o HDFS tira proveito da redundância espacial dos blocos para suprir altas demandas de acesso. Como o HDFS segue uma estratégia *rack-awareness*, é possível identificar o *rack* mais próximo do cliente que possua a réplica requisitada, diminuindo o tempo gasto com o tráfego de dados [1].

Desta forma, a PPR garante um bom equilíbrio em relação [2]: (i) à TF e à disponibilidade dos dados mesmo em caso de falha de um *rack* inteiro; (ii) ao desempenho na escrita dos blocos, já que, na maioria dos casos, a largura de banda entre nodos de um mesmo *rack* é maior do que a largura de banda inter-*rack* e a PPR, ao armazenar mais de uma réplica em um mesmo *rack*, faz com que os dados trafeguem por um menor número de comutadores (*network switches*); e (iii) ao desempenho na leitura dos blocos devido a maior disponibilidade dos dados (operações de leitura podem ser realizadas utilizando a largura de banda de *racks* distintos), assim tornando o acesso aos dados mais rápido.

Embora aumente a disponibilidade do sistema em caso de falhas (redundância de dados em *racks* distintos) e contribua com uma melhor utilização dos recursos computacionais do *cluster* (operações de E/S podem tirar proveito da largura de banda de múltiplos *racks*), a PPR não distribui os blocos de forma igualitária entre os DNs [1].

III. Balanceamento de Réplicas

O HDFS foi projetado para operar sobre um volume massivo de dados. O armazenamento de arquivos maiores resulta em uma maior quantidade de blocos, os quais devem ser posicionados pelo NN. A PPR garante um balanceamento mínimo (réplicas de um mesmo bloco não recaem em um mesmo DN), porém não é suficiente para manter o *cluster* balanceado [3].

A PPR pode favorecer o desbalanceamento de réplicas em dois sentidos. Ao selecionar um *rack* para manter 2/3 das réplicas de um determinado bloco (considerando o FR padrão), tende-se a favorecer o desequilíbrio inter-*rack*. Já o desbalanceamento intra-*rack* (inter-DN) é resultante da arbitrariedade na escolha dos nodos. Em geral, qualquer DN que satisfaça as restrições impostas pela PPR é eletivo ao armazenamento da réplica, sendo a decisão final realizada pelo NN [2].

Além disso, outros aspectos podem contribuir com o desbalanceamento de réplicas no HDFS, tais como: (i) a adição de um novo nodo ao *cluster*, já que este irá competir igualmente com outros DNs para o recebimento dos blocos replicados, resultando em um período de subutilização significativo [4]; (ii) o processo de re-replicação, que está sujeito à mesma política da replicação inicial; e (iii) o comportamento da aplicação do cliente que, caso execute diretamente em um DN do *cluster*, faz com que este, de acordo com a PPR, armazene sempre uma das réplicas localmente.

Sendo um sistema baseado no modelo de acesso WORM (*write once*, *ready many*), muitos dos esforços do HDFS são voltados a maximizar a vazão durante operações de leitura dos dados [2]. O posicionamento dos blocos no *cluster* é um fator crítico para a disponibilidade dos dados e para o desempenho de aplicações de E/S [4]. Um bom posicionamento das réplicas permite reduzir tráfego de dados entre *switches*, o que pode se tornar um gargalo em sistemas de computação intensiva [5].

Um dos princípios do Hadoop é mover a aplicação até os dados, evitando mover os dados propriamente ditos [1]. Trazer a computação para perto dos dados, base do processamento do Hadoop, é conhecido como localidade dos dados (*data locality*) [2]. Esta funcionalidade permite aprimorar a eficiência da plataforma no processamento de grandes *datasets*, já que, por ser local, o acesso aos blocos torna-se mais rápido e menos custoso em termos de consumo de banda.

Como cada bloco é replicado, por padrão, em três DNs diferentes, a probabilidade de que uma tarefa *map* consiga processar a maioria dos blocos localmente é alta [5]. Entretanto, uma distribuição de réplicas desequilibrada tende a afetar a localidade das tarefas *map*[1] do MapReduce [2], podendo ocasionar um maior número de transferências intra-*rack* ou *off-rack* e, assim, consumindo a largura de banda do *cluster*. Além disso, o desbalanceamento pode acarretar em sobrecarga para os DNs com maior utilização do *cluster* (nodos com mais blocos armazenados), prejudicando ainda mais o desempenho na execução de aplicações I/O *bound*.

IV. Trabalhos Relacionados

Estudos passados analisaram a influência da replicação e da re-replicação de blocos no balanceamento de réplicas do HDFS [6]. Ao atestar que o desequilíbrio na distribuição dos dados é uma realidade presente no sistema de arquivos do Hadoop, conduziu-se uma investigação experimental acerca da efetividade da PPR, evidenciando os requisitos necessários para um posicionamento de réplicas eficiente [3].

Uma forma de promover o balanceamento de réplicas no HDFS é agir no momento da distribuição inicial dos blocos. Com isso, é possível impedir – ou reduzir as chances – que o *cluster* fique desequilibrado. Muitas das soluções existentes na literatura [7] [8], envolvem a criação de novas políticas de posicionamento de réplicas que, proativamente, de forma direta ou indireta, contribuem com o balanceamento.

Outra abordagem para endereçar o balanceamento é através de estratégias reativas, onde disparam-se ações corretivas a fim de tornar o posicionamento dos dados entre os nodos mais homogêneo. Para tal, os blocos já armazenados no sistema de arquivos são redistribuídos dentro do *cluster*, visando promover o balanceamento inter-*rack* e/ou entre DNs.

Exemplos dessa abordagem incluem [9], onde é proposto um algoritmo aprimorado para o balanceamento entre *racks* com base em prioridade. A estratégia atua primariamente em

[1] As tarefas *reduce* não tiram vantagem da localidade dos dados [2]. Em geral, a entrada de uma tarefa de redução é a saída de todas as tarefas de mapeamento previamente processadas, que são transferidas, em totalidade, para o local onde a função de redução definida pelo usuário é executada.

equilibrar *racks* sobrecarregados, com isso reduzindo as chances de falha total de *rack* devido à sobrecarga e contribuindo com uma distribuição mais uniforme dos dados. Em [8], é introduzido um algoritmo modificado (*LatencyBalancer*) como parte do balanceador `Tula` que, além da utilização dos DNs, considera variações na latência de escrita e leitura dos discos de armazenamento dos nodos para a realocação dos dados no HDFS. Com isso, os DNs que apresentarem menor latência de disco recebem um número maior de blocos. Os resultados com esta estratégia demostraram reduções de cerca de 20% no tempo de execução de aplicações MapReduce.

Sabendo que o processo de balanceamento pode ser trabalhoso em função do estado em que o *cluster* se encontra, em [10] otimiza-se a redistribuição das réplicas a partir de diferenças de *hardware* dos nodos. Em contraste com o balanceador `Tula`, que não considera variações nos recursos de processamento e memória dos nodos, o algoritmo de balanceamento de [10] baseia-se na capacidade de computação dos DNs. Sendo voltado a instâncias do Hadoop executando em ambientes heterogêneos, os blocos são redistribuídos apenas para DNs específicos, determinados a partir de uma classificação inicial pela heterogeneidade e desempenho de cada nodo. Esta restrição é utilizada como uma estratégia para reduzir o tempo gasto com a transferência dos dados.

Outra solução reativa para o balanceamento de réplicas é o `HDFS Balancer` [4]: ferramenta responsável pela análise do posicionamento dos blocos armazenados no sistema de arquivos para posterior redistribuição de dados entre os DNs. Por ser a base para o desenvolvimento deste trabalho, a Seção IV-A detalha o funcionamento do balanceador do HDFS.

A. HDFS Balancer

O `HDFS Balancer` é uma ferramenta integrada na distribuição do Hadoop que visa o balanceamento de réplicas entre os dispositivos de armazenamento do HDFS. A partir de sua política de execução padrão, o balanceador opera iterativamente movimentando blocos de DNs que apresentarem uma alta utilização (origem) para DNs que possuírem um menor volume de dados armazenado (destino) [11]. Sua execução é disparada sob demanda pelo administrador do *cluster*.

Durante a execução do balanceador, os dispositivos de armazenamento dos DNs são divididos em grupos de acordo com seus tipos (e.g. disco rígido e SSD). Sendo i um DN qualquer e t um tipo de dispositivo de armazenamento, considera-se: (i) $G_{i,t}$ como o grupo de dispositivos de armazenamento do tipo t do DN i; (ii) $U_{i,t}$ como a porcentagem representando a utilização do grupo dos dispositivos do tipo t do DN i; e (iii) $U_{\mu,t}$ como a porcentagem representando a média de utilização de todos os dispositivos do tipo t do *cluster*.

A operação de balanceamento é guiada por um *threshold*, que é passado como parâmetro para a execução do balanceador. Representado como uma porcentagem no intervalo de 0% a 100%, o *threshold* limita a diferença máxima que a $U_{i,t}$ de um $G_{i,t}$ e a utilização geral do *cluster* ($U_{\mu,t}$) pode assumir [2]. Quando a utilização de cada grupo estiver dentro desse limite, o *cluster* é tido como balanceado.

Por exemplo, considerando o *threshold* padrão de 10% e, supondo que o *cluster* esteja com metade de sua capacidade ocupada ($U_{\mu,t}$ em 50%), a ferramenta irá executar até que todos os dispositivos de armazenamento de todos os DNs estejam com utilização entre 40% e 60%. Ao reduzir o *threshold* aumenta-se o equilíbrio do *cluster*, todavia maior será o esforço – em termos de processamento e de transferência de dados – necessário para realizar o balanceamento.

Uma iteração de balanceamento parte da inicialização de um *dispatcher* para a movimentação dos blocos e da obtenção de todos os *DataNodeStorageReports* dos DNs ativos do *cluster*. A partir disto, três ações principais são realizadas [11]: (i) classificação dos $G_{i,t}$ dos DNs, efetuada pelo método *Balancer.init*; (ii) pareamento dos grupos origem-destino, exercido pelo método *chooseStorageGroups*; e (iii) criação de uma *thread* responsável por decidir os blocos a serem movimentados e efetivar a transferência através de um método do *dispatcher*.

O método *Balancer.init* estabelece, para cada $G_{i,t}$, o número máximo de *bytes* que pode ser movimentado na iteração. A partir disto, realizam-se os cálculos necessários para a definição da variável *maxSize2Move*, que equivale ao volume de dados (em *bytes*) necessário para levar a $U_{i,t}$ até a $U_{\mu,t}$.

Em seguida, cada $G_{i,t}$ é classificado em [11]: (i) superutilizado ($U_{i,t} > U_{\mu,t} + \mathit{threshold}$); (ii) acima da média ($U_{\mu,t} + \mathit{threshold} \geq U_{i,t} > U_{\mu,t}$); (iii) abaixo da média ($U_{\mu,t} \geq U_{i,t} \geq U_{\mu,t} - \mathit{threshold}$); ou (iv) subutilizado ($U_{\mu,t} - \mathit{threshold} > U_{i,t}$). Os grupos são então adicionados em listas globais de acordo com as suas classificações.

Após, o método *chooseStorageGroups* determina todos os nodos que irão participar da iteração. Cada $G_{i,t}$ superutilizado (origem) é pareado com um ou mais $G_{i,t}$ subutilizados (destino) em uma relação $1 - N$. Se algum grupo superutilizado possuir *maxSize2Move* satisfeito, ele é removido da lista e não será mais pareado na iteração corrente. Para os grupos superutilizados remanescentes, são selecionados candidatos nos $G_{i,t}$ classificados como abaixo da média (destino). Se ainda houver algum $G_{i,t}$ subutilizado, procuram-se candidatos entre os $G_{i,t}$ acima da média restantes (origem).

Com o pareamento finalizado, o método *dispatchAndCheckContinue* inicia uma *thread* para a seleção e a transferência dos blocos entre os grupos. Quando as movimentações forem concluídas, a iteração é encerrada e o resultado preliminar do balanceamento é registrado. Por fim, todas as listas e mapeamentos são resetados para as iterações futuras, que serão executadas caso o *cluster* ainda não esteja balanceado.

V. Política de Balanceamento Customizada

A operação do `HDFS Balancer` é guiada por uma política de balanceamento, porém, por operar de forma generalizada, a política atual nem sempre é satisfatória e otimizada para os diferentes contextos e cenários de uso em que o HDFS é empregado. Através da compreensão do algoritmo de funcionamento da ferramenta e da revisão do estado da arte em balanceamento de dados em sistemas distribuídos, foi elencado um conjunto de possíveis otimizações e novas funcionalidades para o balanceador nativo do HDFS [12].

As modificações propostas foram estruturadas na forma de uma política de balanceamento de réplicas customizada, que faz uso de um sistema de prioridades para realizar a redistribuição dos blocos com base na topologia do *cluster* e em diferentes métricas de usabilidade do sistema. Com isso, permite-se que o balanceamento atenda demandas de uso específicas de aplicações reais, seja através de otimizações na execução do Hadoop em ambientes heterogêneos ou em explorar potenciais ganhos de confiabilidade e disponibilidade durante a movimentação dos dados.

As prioridades da política customizada foram agrupadas em categorias de acordo com suas características de funcionamento, conforme mostra a Tabela I [13]. A categoria "capacidade dos nodos" realiza a priorização com base em diferenças de *hardware* dos DNs, enquanto a categoria "distribuição dos dados" visa o aumento da disponibilidade dos blocos. Já as prioridades da categoria "estado dos nodos", fazem uso de métricas recuperadas em tempo de execução para reduzir o *overhead* ocasionado pelo processo de balanceamento.

Tabela I
SISTEMA DE PRIORIDADES IMPLEMENTADO PELA POLÍTICA DE BALANCEAMENTO DE RÉPLICAS CUSTOMIZADA.

Categoria	Prioridade
Capacidade dos nodos	Capacidade de armazenamento Capacidade de processamento
Distribuição dos dados	Disponibilidade dos dados
Estado dos nodos	Utilização dos nodos Classificação dos nodos Carga dos nodos
Estado dos *racks*	**Confiabilidade dos *racks*** Utilização dos *racks*

Este trabalho enfoca uma das prioridades da categoria "estado dos *racks*", denominada **confiabilidade dos racks**, que visa customizar a política padrão do `HDFS Balancer` com base em características dos *racks* que agrupam os DNs do HDFS. Na Seção V-A a prioridade em questão é detalhada.

A. Confiabilidade dos Racks

A prioridade de confiabilidade dos *racks* avalia os *racks* do sistema com base na suscetibilidade a falhas de seus DNs. Assim, antes de selecionar um *rack* para o recebimento dos blocos, realiza-se uma verificação da quantidade de DNs inativos no *cluster*, esforçando-se em armazenar uma maior quantidade de dados em *racks* com uma menor ocorrência de falhas. Embora DNs inativos sejam resultado tanto de falhas de *hardware* quanto de *software*, *racks* que possuírem uma proporção elevada entre DNs inativos em relação aos ativos podem estar passando por um período de sobrecarga.

Na política padrão do balanceador apenas os DNs ativos são levados em consideração, porém para a implementação dessa prioridade os DNs inativos também precisam ser considerados. Dessa forma, foi adicionado o método *getDeadDatanodeStorageReport* à classe *NameNodeConnector*, que retorna a lista dos *reports* dos DNs inativos do *cluster*. Esse método é similar ao método *getLiveDatanodeStorageReport* (já existente no código original do `HDFS Balancer`), diferindo apenas no valor da enumeração *DatanodeReportType*, que é repassado ao método *getDatanodeStorageReport* da classe *ClientProtocol* (o método padrão utiliza a enumeração com o valor "*LIVE*", o novo método utiliza o valor "*DEAD*").

Após, adicionou-se um método na classe *Dispatcher* dedicado à coleta dos *reports* de modo similar ao processo já realizado pelo método *Dispatcher.init*, porém consumindo a lista dos DNs inativos ao invés dos ativos. A modificação mais expressiva, por sua vez, se deu no arquivo principal *Balancer*.

Inicialmente, implementou-se um método destinado ao controle e ao mapeamento da quantidade de DNs (ativos e inativos) do *cluster* e seus respectivos *racks*. A Figura 1 exibe o algoritmo do método *mapRacksAndDatanodes*, onde são preenchidas as estruturas *liveDatanodeMap* (L. 2 a 9) e *deadDatanodeMap* (L. 10 a 17). Estas estruturas são consumidas pelo método *computeRFactor*, responsável por quantificar a confiabilidade dos *racks* de acordo com a suscetibilidade a falhas de seus DNs. A chamada desse método foi incorporada em um momento anterior à chamada do método *Balancer.init*.

```
 1: procedure MAPRACKSANDDATANODES
 2:   for each r ∈ datanodeStorageReports do
 3:     dnRackName ← r.getDatanodeInfo().getNetworkLocation()
 4:     if dnRackName ⊄ liveDatanodeMap then
 5:       liveDatanodeMap.put(dnRackName, 0)
 6:     end if
 7:     numliveDNs ← liveDatanodeMap.get(dnRackName)
 8:     liveDatanodeMap.put(dnRackName, numLiveDNs + 1)
 9:   end for
10:   for each r ∈ dispatcher.getDeadDNStorageReports()) do
11:     dnRackName ← r.getDatanodeInfo().getNetworkLocation()
12:     if dnRackName ⊄ deadDatanodeMap then
13:       deadDatanodeMap.put(dnRackName, 0)
14:     end if
15:     numDeadDNs ← deadDatanodeMap.get(dnRackName)
16:     deadDatanodeMap.put(dnRackName, numDeadDNs + 1)
17:   end for
18: end procedure
```

Figura 1. Método para mapeamento dos DNs ativos e inativos em cada *rack*.

A Figura 2 exibe o algoritmo definido no método *computeRFactor*. Inicialmente, é realizado o cálculo da taxa de falhas de cada um dos *racks* (L. 2 a 10). Se um *rack* não possuir DNs inativos, sua taxa de falhas será zero. Caso contrário, calcula-se o quanto estes representam em relação ao total de DNs do *rack* (L. 7). Em seguida, os valores calculados para cada um dos *racks* do *cluster* (intervalo de 0 a 1) são inseridos na estrutura *failureRateMap* (L. 9).

Após, estima-se um fator de confiança para cada *rack* (*rFactor*). Para tal, aplicou-se a normalização *min-max* nas taxas de falhas dos *racks* (L. 16), assim permitindo a definição de um intervalo controlado de valores. Com a normalização, torna-se o valor mínimo de um conjunto de valores em 0, o valor máximo em 1 e, qualquer outro valor, em um decimal proporcional entre 0 e 1. A Equação 1 aplica a normalização

```
1:  procedure COMPUTERFACTOR
2:    for each (rack, numLiveDNs) ∈ liveDatanodeMap do
3:      failureRate ← 0.0
4:      if rack ∈ deadDatanodeMap then
5:        numDeadDNs ← deadDatanodeMap.get(rack)
6:        totalDNsRack ← numLiveDNs + numDeadDNs
7:        failureRate ← numDeadDNs / totalDNsRack
8:      end if
9:      failureRateMap.put(rack, failureRate)
10:   end for
11:   max ← max(failureRateMap.values())
12:   min ← min(failureRateMap.values())
13:   for each (rack, failureRate) ∈ failureRateMap do
14:     calculatedReliability ← 0.5
15:     if (max − min) ≠ 0.0 then
16:       normalizatedRate ← ((failureRate)−min)/(max−min))
17:       rFactor ← 1.0 − normalizatedRate
18:     end if
19:     rMap.put(rack, reliabilityFactor)
20:   end for
21: end procedure
```

Figura 2. Método responsável por quantificar a confiabilidade dos *racks*.

min-max sobre a taxa de falhas de um *rack* r (T_r) a fim de obter seu respectivo fator de confiança, i.e. *rFactor* (T'_r).

$$T'_r = \frac{T_r - min}{max - min} \quad (1)$$

Caso não haja ocorrência de DNs falhos nos *racks* do *cluster*, *max* terá valor zero e, assim, T'_r passa a ser um valor médio de 0,5 (L. 14), fazendo com que a definição de *maxSize2Move* para os grupos de dispositivos de armazenamento seja similar ao cálculo realizado pela política padrão do `HDFS Balancer` ($|U_{i,t}$ - $U_{\mu,t}|$). Caso contrário, o fator de confiança é definido com base na quantidade de DNs ativos e inativos no *rack* (L. 17). Vale observar que, caso o valor armazenado em *failureRateMap* para um determinado *rack* seja zero, T'_r terá o valor 1 (motivo do uso do valor inverso na linha 17).

Os fatores de confiança dos *racks* são salvos na estrutura *rMap* (L. 19), que será posteriormente acessada pelo método *calcMaxSize2MoveBasedOnRFactor*. Este método, cujo algoritmo é exibido na Figura 3, estabelece o valor de *maxSize2Move* de um $G_{i,t}$ de acordo com o fator de confiança de seu *rack*. Sua chamada foi incorporada ao método *Balancer.init* em um momento anterior à classificação dos grupos.

Inicialmente, para cada $G_{i,t}$, calcula-se a diferença de sua utilização ($U_{i,t}$) para o limite superior ($U_{\mu,t}$ + *threshold*) e para o limite inferior ($U_{\mu,t}$ − *threshold*) que são considerados pelo balanceador (L. 4 e 5). Esses valores, por sua vez, permitem definir a quantidade de *bytes* necessária para levar a $U_{i,t}$ de um grupo até o limite máximo para que este seja considerado como acima ou abaixo da média (L. 6 e 7). Em seguida, o valor de *maxSize2Move* é definido de acordo com a classificação e a T'_r do *rack* ao qual o $G_{i,t}$ pertence.

De forma geral, o valor de *maxSize2Move* para um $G_{i,t}$ de um *rack* com uma T'_r elevada, aproxima-se do volume de *bytes* necessário para elevar sua $U_{i,t}$ até o limite superior. Já para um $G_{i,t}$ de um *rack* com uma T'_r baixa, aproxima-se do

```
1:  procedure CALCMAXSIZE2MOVEBASEDONRFACTOR
2:    utilizationDiff ← utilization − average
3:    thresholdDiff ← |utilizationDiff| − threshold
4:    supLimDiff ← utilization − (average + threshold)
5:    infLimDiff ← utilization − (average − threshold)
6:    bytes2SupLim ← |supLimDiff| × capacity / 100
7:    bytes2InfLim ← |infLimDiff| × capacity / 100
8:    thresholdBytes ← bytes2InfLim + bytes2SupLim
9:    key ← r.getDatanodeInfo().getNetworkLocation()
10:   if utilizationDiff > 0 then                          ▷ origem
11:     if thresholdDiff ≤ 0 then                ▷ G_{i,t} acima da média
12:       rBasedBytes ← thresholdBytes × (1 − rMap.get(key))
13:       maxSize2Move ← max(0, rBasedBytes−bytes2SupLim)
14:     else                                    ▷ G_{i,t} superutilizado
15:       rBasedBytes ← bytes2InfLim × (1 − rMap.get(key))
16:       maxSize2Move ← max(bytes2SupLim, rBasedBytes)
17:     end if
18:   else                                                 ▷ destino
19:     if thresholdDiff ≤ 0 then               ▷ G_{i,t} abaixo da média
20:       rBasedBytes ← thresholdBytes × rMap.get(key)
21:       maxSize2Move ← max(0, rBasedBytes − bytes2InfLim)
22:     else                                     ▷ G_{i,t} subutilizado
23:       rBasedBytes ← bytes2SupLim × rMap.get(key)
24:       maxSize2Move ← max(bytes2InfLim, rBasedBytes)
25:     end if
26:     maxSize2Move ← min(remaining, maxSize2Move)
27:   end if
28:   return min(maxSize2Move, max)                 ▷ max = 10GB
29: end procedure
```

Figura 3. Método dedicado ao cálculo customizado da variável *maxSize2Move* de um $G_{i,t}$ com base no fator de confiança de seu *rack* (T'_r).

volume de *bytes* necessário para reduzir sua $U_{i,t}$ até o limite inferior. Para idealizar este comportamento, é importante observar que, em um grupo origem (superutilizado ou acima da média), *maxSize2Move* determina a redução máxima de sua $U_{i,t}$ (i.e. não é possível aumentar sua utilização para um valor superior à $U_{i,t}$ atual). Analogamente, para um grupo destino (subutilizado ou abaixo da média), *maxSize2Move* determina a extensão máxima de sua $U_{i,t}$ (i.e. não é possível reduzir sua utilização para um valor inferior à $U_{i,t}$ atual).

Sendo assim, se a T'_r do *rack* de um $G_{i,t}$ classificado como acima da média resultar em uma utilização superior à $U_{i,t}$ atual do grupo, *maxSize2Move* deverá ter o valor zero (L. 13), garantindo dessa forma que sua utilização não seja reduzida. Similarmente, se a T'_r do *rack* de um $G_{i,t}$ abaixo da média resultar em uma utilização inferior à $U_{i,t}$ atual do grupo, *maxSize2Move* também deverá ter o valor zero (L. 21), garantindo que sua utilização não seja aumentada.

Além disso, em $G_{i,t}$ superutilizados de *racks* com alta T'_r, o valor de *maxSize2Move* deve, ao mínimo, ser o suficiente para permitir a classificação do grupo como acima da média (garantido pelo método *max* na linha 16). Já em $G_{i,t}$ subutilizados de *racks* com baixa T'_r, o valor de *maxSize2Move* deve ao menos ser suficiente para a classificação do grupo como abaixo da média (garantido pelo método *max* na linha 24).

Por fim, são aplicadas duas validações (definidas também no cálculo realizado pela política padrão do balanceador): (i) em grupos destino (*utilizationDiff* menor que zero), o

valor da variável *maxSize2Move* deve ser limitado pelo espaço de armazenamento restante no $G_{i,t}$ (L. 26); e (ii) *maxSize2Move* deve respeitar o valor de *max* (dado pela propriedade *dfs.balancer.max-size-to-move*) (L. 28).

VI. Experimentação e Discussão

A prioridade de confiabilidade dos *racks* foi avaliada através de um experimento realizado na plataforma GRID'5000, com o Hadoop (versão 2.9.2) operando em modo totalmente distribuído, executando sobre uma distribuição Debian 9.9. Todos os nodos configurados pertenciam ao *cluster suno*, localizado no *site Sophia*, cada um com 2 processadores Intel Xeon E5520 (4 *cores* por CPU), 32GB de memória RAM, 508,54GB de armazenamento HDD e conexão *Gigabit Ethernet*.

A carga de dados foi realizada pelo `TestDFSIO` [2] (versão 1.8): um *benchmark* distribuído proprietário do Hadoop, que permite medir o desempenho do HDFS através da execução de tarefas MapReduce focadas em E/S intensiva. Foram escritos 10 arquivos de 30GB cada um com FR padrão de 3 réplicas por bloco, totalizando aproximadamente 908,55GB de dados no sistema. Para que a suscetibilidade a falhas dos *racks* fosse quantificada pela prioridade, falhas de DN foram induzidas durante a escrita dos arquivos pelo *benchmark*.

O cenário de testes considerou 15 DNs dispostos em 3 *racks* distintos (R_1, R_2, R_3), cada um mantendo 3, 5 e 7 DNs. A introdução das falhas foi realizada pelo comando *kill* do Linux aos processos dos DNs selecionados. Ao total, foram induzidas 2 falhas de DN no *rack* R_2 e 4 falhas no *rack* R_3. Com isso, após as falhas, cada *rack* manteve exatamente 3 DNs ativos. Os DNs para indução das falhas foram escolhidos arbitrariamente.

As falhas são identificadas pelo NN em decorrência da ausência de mensagens *heartbeat* em um intervalo pré-definido. Como o tempo para o NN marcar um DN como inativo é relativamente longo, este período foi reduzido (através de um conjunto de parâmetros de configuração do HDFS) para aproximadamente 20 segundos. O tempo para a ocorrência da primeira falha de DN e o intervalo entre as falhas foi fixado em 60 segundos. Desta forma, o processo de re-replicação é disparado pelo NN e efetivamente concluído antes da escrita dos arquivos ser finalizada pelo `TestDFSIO`.

A capacidade de armazenamento total do sistema, considerando apenas os DNs ativos, era de 4,47TB ($U_{\mu,DISK}$ em 20,95%). A Tabela II exibe a ocupação de cada DN ativo, em GB ($O_{i,DISK}$), e sua respectiva porcentagem de utilização ($U_{i,DISK}$). O fator de confiança de cada *rack* (T'_r) foi calculado de acordo com a normalização *min-max* baseada na quantidade de DNs inativos no *rack* no momento do balanceamento. Com isso, dada a quantidade de falhas de DN induzidas em cada *rack*, $T'_{R1} > T'_{R2} > T'_{R3}$.

A utilização total dos *racks* (proporção da ocupação acumulada para a capacidade total de seus DNs) antes do balanceamento era de, respectivamente, 58,76%, 60,66% e 59,24%. Sendo assim, o *rack* com a maior T'_r (R_1) estava mantendo um volume de dados menor que os demais *racks*, os quais apresentavam incidência de falhas de DN (R_2 com 2 DNs inativos e R_3 com 4 DNs inativos). Após o balanceamento

Tabela II
Estado do HDFS antes e após o balanceamento de réplicas com a prioridade de confiabilidade dos racks.

Rack	T'_r	DataNode	s/ balanceamento		c/ balanceamento	
			$O_{i,DISK}$ (GB)	$U_{i,DISK}$ (%)	$O_{i,DISK}$ (GB)	$U_{i,DISK}$ (%)
R_1	1,0	DN_{01}	78,37	15,41	123,52	24,29
		DN_{02}	135,51	26,65	126,23	24,82
		DN_{03}	84,92	16,70	126,01	24,78
R_2	0,3	DN_{04}	96,16	18,91	95,75	18,83
		DN_{05}	91,20	17,93	102,30	20,12
		DN_{06}	121,13	23,82	98,17	19,30
R_3	0,0	DN_{07}	75,83	14,91	75,97	14,94
		DN_{08}	149,50	29,40	85,13	16,74
		DN_{09}	75,93	14,93	75,97	14,94

com base na confiança dos *racks*, a utilização de R_1, R_2 e R_3 foi levada para, respectivamente, 73,89%, 58,25% e 46,62%. Sendo assim, observa-se que, estando de acordo com a confiabilidade calculada para cada *rack* do *cluster* (i.e. fatores de confiança), $U_{R1,DISK} > U_{R2,DISK} > U_{R3,DISK}$.

De forma a avaliar o impacto do balanceamento com a estratégia de priorização proposta no desempenho do sistema, foram realizadas 20 execuções distintas do `TestDFSIO` voltadas à leitura dos dados armazenados no HDFS. A análise por meio desse *benchmark* mostra-se interessante pois, através de seu comportamento característico *I/O bound*, possibilita-se a investigação de possíveis melhorias na localidade espacial dos dados impulsionadas pelo equilíbrio das réplicas no *cluster*.

A Figura 4 exibe os tempos para leitura dos dados no HDFS. Com o posicionamento inicial dos blocos seguindo a PPR, a média dos tempos foi de 778,71s. Após o balanceamento, esse valor foi reduzido para 584,61s. Considerando a variação percentual dada por $((T_b - T_a) \;/\; T_a \times 100)$, onde T_a e T_b equivalem, respectivamente, às médias aritméticas dos tempos das 20 execuções do *benchmark* `TestDFSIO` antes e após a execução do `HDFS Balancer` com a prioridade de confiabilidade dos *racks*, a variação alcançada foi de $-24{,}92\%$, indicando a redução obtida no tempo de leitura dos dados.

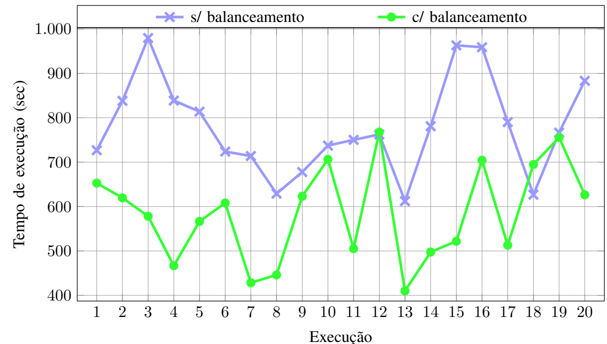

Figura 4. Tempo para leitura dos dados antes e após o balanceamento.

Com a capacidade do HDFS em processar conjuntos de tarefas paralelas e independentes sendo melhor explorada em um *cluster* equilibrado, outras métricas tornam-se passíveis

de otimização. Na Figura 5, são exibidas as taxas de E/S alcançadas pelo *benchmark*. Essa taxa relaciona a velocidade de transferência média obtida por cada tarefa pela quantidade total de tarefas *map* executadas. Para o `TestDFSIO`, a quantidade de tarefas de mapeamento geradas equivale, por padrão, à quantidade de arquivos manipulados (no experimento, a leitura foi realizada sobre 10 arquivos). A taxa média de E/S foi de 47,33MB/s sem o balanceamento e de 57,82MB/s após o balanceamento com base na confiança dos *racks*. A variação percentual, considerando os valores médios das 20 execuções, foi de 22,16%, indicando aumento na taxa de transferência.

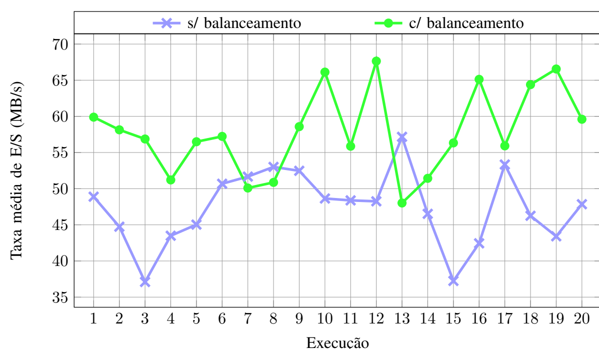

Figura 5. Taxa de transferência de dados antes e após o balanceamento.

A Figura 6, por sua vez, exibe o *throughput* obtido durante a leitura dos dados antes e após o equilíbrio do *cluster*. O *throughput* é dado pela razão do volume total de dados processados (em MB) pela soma dos tempos (em segundos) gastos por cada tarefa (devido ao paralelismo, este valor é superior ao tempo total de execução). O *throughput* médio foi de 45,7MB/s sem o balanceamento e de 57,28MB/s com o uso do balanceador. A variação percentual alcançada foi de 25,36%, indicando aumento no *througput* da aplicação após o balanceamento com a prioridade de confiabilidade dos *racks*.

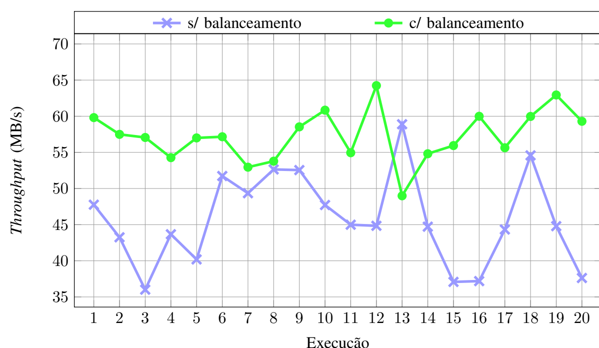

Figura 6. *Throughput* de leitura antes e após o balanceamento.

VII. Considerações Finais

O `HDFS Balancer` é uma solução nativa do Hadoop voltada ao balanceamento de réplicas. Por operar de forma generalizada, sua política de balanceamento atual não considera possíveis demandas de disponibilidade e confiabilidade durante a redistribuição dos blocos entre os *racks* do *cluster*.

Esse trabalho apresentou uma estratégia de balanceamento customizada para o `HDFS Balancer` que prioriza *racks* com alta confiabilidade durante a movimentação das réplicas. Para tal, o fator de confiança de cada *rack* foi estimado com base na taxa de falhas de seus `DataNodes`. Após detalhar a implementação, conduziu-se uma investigação experimental que – além de validar a proposta – permitiu demonstrar que a execução do balanceador priorizando *racks* com menor ocorrência de falhas continua a proporcionar melhorias de desempenho significativas após o balanceamento de réplicas.

Trabalhos futuros envolvem avaliar a solução proposta em cenários com indução de falhas transientes em nodos e com testes de estresse considerando falha total de *rack*. Além disso, pretende-se associar a prioridade de confiabilidade dos *racks* apresentada neste trabalho com prioridades voltadas à otimização da disponibilidade dos dados no HDFS.

Agradecimentos

Os experimentos apresentados neste trabalho foram conduzidos na plataforma Grid'5000, apoiada por um grupo de interesses científicos hospedado por Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais detalhes em https://www.grid5000.fr).

Referências

[1] Apache Software Foundation. (2018) Apache hadoop. [Online]. Available: https://hadoop.apache.org/docs/r2.9.2/. [Acesso: 19 de Maio, 2019].

[2] T. White, *Hadoop: The Definitive Guide*, 4th ed. Sebastopol: O'Reilly Media, Inc., 2015.

[3] R. W. A. Fazul and P. P. Barcelos, "Efetividade da política de posicionamento de blocos no balanceamento de réplicas do hdfs," in *Anais do XX Workshop de Testes e Tolerância a Falhas*. SBC, 2019, pp. 79–92.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Symposium on Mass Storage Systems and Technologies*. Incline Village: IEEE, 2010, pp. 1–10.

[5] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. Ottawa: IEEE Computer Society, 2012, pp. 419–426.

[6] R. W. A. Fazul, P. V. Cardoso, and P. P. Barcelos, "Análise do impacto da replicação de dados implementada pelo apache hadoop no balanceamento de carga," in *Anais do X Computer on the Beach*. Florianópolis: Universidade do Vale do Itajaí, 2019, pp. 579–588.

[7] C. B. VishnuVardhan and P. K. Baruah, "Improving the performance of heterogeneous hadoop cluster," in *Fourth International Conference on Parallel, Distributed and Grid Computing*. IEEE, 2016, pp. 225–230.

[8] J. Dharanipragada, S. Padala, B. Kammili, and V. Kumar, "Tula: A disk latency aware balancing and block placement strategy for hadoop," in *International Conference on Big Data*. IEEE, 2017, pp. 2853–2858.

[9] K. Liu, G. Xu, and J. Yuan, "An improved hadoop data load balancing algorithm," *Journal of Networks*, vol. 8, no. 12, pp. 2816–2822, 2013.

[10] A. Shah and M. Padole, "Load balancing through block rearrangement policy for hadoop heterogeneous cluster," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Bangalore: IEEE, 2018, pp. 230–236.

[11] Hortonworks Data Plataform. (2018) Scaling namespaces and optimizing data storage. [Online]. Available: https://docs.hortonworks.com/HDPDocuments/HDP3/HDP-3.1.0/data-storage/content/balancing_data_across_hdfs_cluster.html. [Acesso: 2 de Junho, 2019].

[12] R. W. A. Fazul and P. P. Barcelos, "Política customizada de balanceamento de réplicas para o hdfs balancer do apache hadoop," in *Anais do XX Workshop de Testes e Tolerância a Falhas*. SBC, 2019, pp. 93–106.

[13] R. W. A. Fazul. (2019) Implementação de uma política customizada de balanceamento de réplicas para o hdfs balancer do apache hadoop. [Online]. Disponível em: https://repositorio.ufsm.br/. Universidade Federal de Santa Maria, Centro de Tecnologia. (no prelo).