Priorização no Balanceamento de Réplicas em Instâncias Heterogêneas do HDFS baseada na Capacidade dos Nodos

Rhauani Weber Aita Fazul[1], **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 – RS – Brasil

{rwfazul, pitthan}@inf.ufsm.br

Abstract. *The HDFS Balancer is an Apache Hadoop daemon that redistributes blocks by moving them across the cluster until the utilization of every node is within a certain threshold. However, the balancer tool is not optimized for HDFS instances running in heterogeneous environments. This work presents a customized balancing policy for HDFS Balancer that allows data distribution based on differences in the capacities of the nodes. Thus, nodes with high capacity are prioritized to receive a high number of replicas during block rearrangement.*

Resumo. *O HDFS Balancer é o balanceador de réplicas nativo do Apache Hadoop, que opera em redistribuir os blocos de dados armazenados no sistema até que a utilização de todos os nodos do cluster fique dentro de um determinado threshold. Entretanto, a ferramenta não é otimizada para o balanceamento de instâncias do Hadoop executando em ambientes heterogêneos. Esse trabalho apresenta uma customização na política de operação padrão do HDFS Balancer que faz com que o reposicionamento das réplicas seja realizado considerando diferenças nas capacidades dos nodos. Dessa forma, nodos com alta capacidade são priorizados para o recebimento de um maior volume de dados.*

1. Introdução

O HDFS (*Hadoop Distributed File System*) é o sistema de arquivos distribuído do Apache Hadoaop[1], projetado para ser tolerante a falhas mesmo quando executa em *clusters* com *hardware* comum (i.e. não confiável). Um *cluster* HDFS baseia-se em uma arquitetura mestre-escravo formada por dois tipos de nós: `NameNode` (NN) e `DataNode` (DN) [Foundation 2018]. O NN é o servidor mestre que gerencia o *namespace* e os metadados do sistema e, também, controla o acesso e a distribuição dos arquivos. Enquanto isso, os DNs são os *workers* que recuperam e armazenam efetivamente os dados.

O HDFS adota uma estrutura de armazenamento própria, otimizada para cenários de *big data*. Ao ser inserido no sistema, os arquivos são automaticamente segmentados em blocos de dados de tamanho fixo (por padrão 128MB). Ao executar em equipamentos de baixo custo, as chances de falhas de nodo em um *cluster* HDFS são elevadas [White 2015]. Mesmo com esta condição, o sistema deve garantir alta disponibilidade e confiabilidade, de modo que nenhum bloco seja perdido. Para tal, o HDFS faz uso da replicação de dados: um mecanismo de tolerância a falhas baseado em redundância.

[1]`https://hadoop.apache.org/`

A replicação é essencial para assegurar a integridade e a disponibilidade dos blocos armazenados no sistema, além de ser utilizada como uma estratégia para suprir altas demandas de acesso. A forma de distribuição das réplicas entre os nodos, entretanto, influencia diretamente no balanceamento no *cluster*. Quando os DNs armazenam quantidades desproporcionais de dados, o desempenho do HDFS é afetado [Fazul et al. 2019]. Uma forma de equilibrar a distribuição de dados no sistema é através do `HDFS Balancer`, o balanceador nativo do Hadoop. Todavia, a política de operação padrão da ferramenta não é otimizada para instâncias do HDFS executando em ambientes heterogêneos.

Este trabalho apresenta uma customização para o `HDFS Balancer` sob a forma de uma prioridade de balanceamento. Com a solução proposta, diferenças nas capacidades de armazenamento ou de processamento dos nodos passam a ser consideradas pelo balanceador durante a redistribuição dos dados. Dessa forma, DNs que apresentarem maior capacidade tornam-se prioritários ao recebimento controlado de um maior número de réplicas, enquanto DNs com menor capacidade ficam responsáveis por armazenar um menor volume de dados. Adicionalmente, uma investigação experimental foi conduzida de forma a validar a implementação e analisar possíveis melhorias na localidade dos dados impulsionadas pelo balanceamento de réplicas com base na capacidade dos nodos.

O artigo está organizado em seis seções. A Seção 2 é voltada ao processo de replicação de dados no HDFS. A Seção 3 dedica-se ao balanceamento de réplicas e aos trabalhos relacionados. A Seção 4 detalha a solução desenvolvida. A Seção 5 apresenta e discute os resultados obtidos. Por fim, a Seção 6 aponta as considerações finais.

2. Replicação de Blocos no HDFS

A replicação é o principal mecanismo de tolerância a falhas (TF) do HDFS. O mecanismo consiste na criação de cópias dos blocos armazenados no sistema, visando o aumento da confiabilidade e da disponibilidade dos dados por meio de redundância. As réplicas são armazenadas em diferentes DNs, de forma que, em caso de falha de um nodo, os blocos comprometidos possam ser acessados a partir de um ou mais DNs que contenham suas réplicas. A quantidade de réplicas de cada bloco é definida pelo Fator de Replicação (FR).

Além da distribuição inicial dos blocos, é necessário que o NN monitore ativamente o estado das réplicas do sistema, disparando, quando preciso, o processo de re-replicação. A re-replicação é fundamental para a manutenção da confiabilidade do HDFS e, dentre outros motivos, pode ser originada por falhas de DN [Foundation 2018]. Um DN inativo é identificado pelo NN através da ausência de mensagens *Hearbeat* em um intervalo pré-definido. A partir disto, o FR dos blocos armazenados no DN em questão é decrementado e a re-replicação pode ser disparada pelo NN. Ambos os processos de replicação e re-replicação são realizados de forma transparente pelo HDFS e necessitam de uma decisão do NN: a escolha dos DNs para o armazenamento das réplicas.

Esta escolha deve assegurar a disponibilidade dos blocos em caso de falhas e otimizar o desempenho do HDFS em operações sobre os dados. Para tal, o NN é guiado por uma Política de Posicionamento de Réplicas (PPR), que faz uso de uma estratégia *rack-aware* visando aprimorar a confiabilidade e a TF do sistema com base na arquitetura do *cluster* [Foundation 2018]. Para um FR padrão de três réplicas por bloco, a PPR faz com que a primeira réplica seja armazenada no DN local (se o cliente HDFS estiver executando fora do *cluster* o DN é escolhido aleatoriamente) e que as duas réplicas seguintes sejam

armazenadas em um *rack* remoto diferente do *rack* da primeira réplica, porém em dois DNs distintos [White 2015]. Em caso de um FR maior, os DNs adicionais são escolhidos arbitrariamente, porém evitando armazenar muitas réplicas em um mesmo *rack*.

Para reduzir a sobrecarga de armazenamento das múltiplas réplicas de um mesmo bloco, o HDFS usa um *pipeline* de replicação [Achari 2015], onde os DNs podem, simultaneamente, receber e encaminhar dados. Assim, permite-se que o processo de escrita seja otimizado e transparente ao cliente, que precisa interagir com apenas um único nodo. Já durante a leitura, o HDFS tira proveito da replicação para suprir altas demandas de acesso. Para tal, as réplicas mais próximas da origem da solicitação são preferidas sobre as réplicas remotas, diminuindo o tempo e o custo gastos com o tráfego de dados. Embora a redundância dos dados em *racks* distintos aumente a confiabilidade do sistema e permita uma melhor utilização dos recursos computacionais durante operações de entrada e saída (E/S), a PPR não distribui os blocos de forma igualitária entre os DNs [Foundation 2018].

3. Balanceamento de Réplicas

Um dos pilares do Hadoop é mover as tarefas de computação para onde estão as réplicas, evitando mover os dados em si e, assim, utilizando o processamento local como forma de evitar o consumo da largura de banda do *cluster*. Esta funcionalidade é conhecida como *data locality* [White 2015]. Em um *cluster* balanceado, é possível explorar a localidade dos dados como forma de aprimorar o desempenho de aplicações voltadas a E/S [Fazul et al. 2019]. Porém, a medida que o desbalanceamento de réplicas se agrava no HDFS, a localidade dos blocos é afetada, podendo ocasionar um maior tráfego *off-rack* e fazer com que os recursos computacionais não sejam utilizados de forma otimizada.

A PPR garante um balanceamento mínimo, porém favorece o desbalanceamento *inter*-nodo (DNs são escolhidos arbitrariamente para manter as réplicas, sem considerar a ocupação de cada nodo) e inter-*rack* (um mesmo *rack* é selecionado para manter 2/3 das réplicas de um determinado bloco). Além disso, outros aspectos podem promover o desequilíbrio no sistema, tais como: (i) o processo de re-replicação, também sujeito à PPR; (ii) a adição de um DN ao *cluster*, já que este irá competir igualmente com os demais DNs para o recebimento dos blocos replicados, resultando em um período de subutilização significativo; e (iii) o comportamento da aplicação do cliente que, em função da PPR, armazena uma das réplicas localmente [Hortonworks 2018]. Na Seção 3.1 são apresentadas possíveis abordagens para promover o balanceamento de réplicas no HDFS.

3.1. Trabalhos Relacionados

Diferentes estratégias para o balanceamento podem ser encontradas na literatura. Em geral, as soluções podem ser classificadas em proativas ou reativas. Uma solução proativa visa manter o equilíbrio no HDFS através de modificações na PPR. Dessa forma, no momento da distribuição inicial dos blocos, o NN passa a considerar o volume de dados armazenado em cada DN. Exemplos dessa abordagem incluem [Ibrahim et al. 2016], em que os DNs são marcados como livres ou ocupados com base em um histórico da distribuição das réplicas no *cluster*. Durante a distribuição das réplicas priorizam-se os *racks* com um maior número de DNs livres. Após, escolhe-se o DN que, estando de acordo com a PPR, possuir o menor espaço de armazenamento ocupado.

Mesmo que as soluções proativas contribuam para manter o HDFS balanceado, nem sempre é possível impedir o desequilíbrio na distribuição de réplicas. Em certas

situações, como a adição de novos nodos no sistema, o reposicionamento das réplicas já armazenadas torna-se necessário. Exemplos de abordagens reativas para o balanceamento incluem [Liu et al. 2013], que apresentam um algoritmo de balanceamento para equilibrar *racks* sobrecarregados, visando reduzir as chances de falha total de *rack* devido à sobrecarga e contribuindo com uma distribuição mais uniforme dos dados. Já [Shah and Padole 2018] focam em otimizar o processo de redistribuição das réplicas aproveitando-se da capacidade de processamento dos nodos, onde os blocos são redistribuídos apenas para DNs específicos, determinados a partir de uma classificação inicial pela heterogeneidade e capacidade de computação de cada DN. Outra possibilidade para o balanceamento reativo é o `HDFS Balancer` [Shvachko et al. 2010]. Por ser a base para o desenvolvimento deste trabalho, a Seção 3.1.1 é dedicada a este balanceador.

3.1.1. HDFS Balancer

O `HDFS Balancer` [Shvachko et al. 2010] é uma ferramenta do Hadoop dedicada ao balanceamento de réplicas entre dispositivos de armazenamento no HDFS. A ferramenta opera em função de um *threshold* (por padrão 10%), que é passado como parâmetro para a execução. Representado como uma porcentagem no intervalo de 0% a 100%, o *threshold* limita a diferença máxima que a utilização dos DNs (proporção do espaço em uso no nodo para a capacidade total do nodo) e a utilização geral do *cluster* (proporção do espaço em uso no *cluster* para a capacidade total do *cluster*) pode assumir [White 2015]. Quando a utilização de cada DN estiver dentro desse limite, o *cluster* é considerado balanceado.

A *daemon* do `HDFS Balancer` é disparada sob demanda pelo administrador do sistema e foi projetada para operar sem afetar as demais aplicações em execução no *cluster* [White 2015]. De todo modo, é possível definir a largura de banda máxima que a ferramenta pode consumir (por padrão 1MB/s). Quanto maior a largura permitida, mais rápido o balanceamento será realizado, porém aumentando as chances de sobrecarga por gerar maior concorrência com os processos do sistema. O fluxo de execução alto-nível do `HDFS Balancer` pode ser dividido em diferentes etapas realizadas sucessivamente durante a operação da ferramenta [Hortonworks 2018], que são apresentadas a seguir.

O balanceamento inicia com a etapa de **classificação dos grupos de dispositivos**, onde os dispositivos de armazenamento dos DNs são agrupados de acordo com seus tipos. Sendo i um DN qualquer e t um tipo de dispositivo de armazenamento, considera-se: (i) $G_{i,t}$ como o grupo de dispositivos do tipo t do DN i; (ii) $U_{i,t}$ como a porcentagem que representa a utilização do grupo dos dispositivos do tipo t do DN i; e (iii) $U_{\mu,t}$ como a porcentagem que representa a média de utilização de todos os dispositivos do tipo t do *cluster*. Os grupos são adicionados em listas globais de acordo com a sua classificação, sendo elas: (i) superutilizado (*over-utilized*), quando $U_{i,t} > U_{\mu,t} + threshold$; (ii) acima da média (*above-average*), quando $U_{i,t} > U_{\mu,t}$ (L. 3); (iii) abaixo da média (*below-average*), quando $U_{\mu,t} \geq U_{i,t} \geq U_{\mu,t} - threshold$; (iv) subutilizado (*under-utilized*), quando $U_{\mu,t} - threshold > U_{i,t}$. A classificação é realizada em um método denominado *Balancer.init*, que também é responsável por definir, para cada $G_{i,t}$, o volume máximo de dados a ser transferido ou recebido em uma iteração de balanceamento.

Este valor é dado pela variável *maxSize2Move*, cujo cálculo é apresentado na Figura 1. Por padrão, *maxSize2Move* equivale ao volume de dados (em *bytes*) necessário

para levar a $U_{i,t}$ até a $U_{\mu,t}$. Para os $G_{i,t}$ tidos como destino (*utilizationDiff* menor que zero), o valor de *maxSize2Move* é limitado pelo espaço de armazenamento restante no grupo (L. 9). Ao final, garante-se que o valor de *maxSize2Move* não extrapole o valor definido na propriedade *dfs.balancer.max-size-to-move* (por padrão 10GB) (L. 11). Se o *cluster* HDFS não possuir nenhum $G_{i,t}$ subutilizado ou superutilizado ele é considerado balanceado. Caso contrário, a execução do `HDFS Balancer` continua na etapa seguinte.

Figura 1. Cálculo padrão para a definição de *maxSize2Move* de cada grupo.

```
 1: utilization ← getUtilization(r, t)                    ▷ utilização do G_{i,t} (U_{i,t})
 2: average ← getAvgUtilization(t)                        ▷ média do cluster (U_{μ,t})
 3: utilizationDiff ← utilization − average
 4: capacity ← getCapacity(r, t)
 5: remaining ← getRemaining(r, t)
 6: thresholdDiff ← |utilizationDiff| − threshold
 7: maxSize2Move ← |utilizationDiff| × capacity / 100     ▷ valor em bytes
 8: if utilizationDiff < 0 then                           ▷ G_{i,t} destino (subutilizado ou abaixo da média)
 9:   maxSize2Move ← min(remaining, maxSize2Move)
10: end if
11: maxSize2Move ← min(max, maxSize2move)                 ▷ max por padrão = 10GB
```

Na etapa de **pareamento dos grupos**, o método *chooseStorageGroups* forma pares origem-destino entre os grupos classificados anteriormente. 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. Se ainda houver algum $G_{i,t}$ subutilizado, procuram-se candidatos entre os $G_{i,t}$ acima da média restantes. A estratégia de pareamento procura, inicialmente, grupos de um mesmo *rack*. Caso não seja possível formar pares no mesmo *rack*, aceitam-se grupos de qualquer *rack*.

No **agendamento de movimentação dos blocos**, o método *dispatchAndCheckContinue* inicia uma *thread* responsável por decidir quais dos blocos da origem devem ser movimentados para o destino. Um bloco do grupo de um DN origem é eletivo ao movimento se: (i) estiver armazenado em um dispositivo do mesmo tipo que o destino; (ii) não estiver em processo de movimentação, nem for um dos blocos já movimentados na iteração; (iii) não possuir uma réplica já existente no destino; e (iv) após a movimentação, continuar em concordância com a PPR. Para diminui o tráfego necessário para a transferência dos dados entre os nodos, o DN mais próximo do destino que possuir uma réplica do bloco a ser movimentado é utilizado como um *proxy*.

Na etapa de **transferência do bloco**, o DN destino copia o bloco mantido no DN *proxy* para seu armazenamento local. Após, ele envia um alerta para o NN, que dispara a deleção da réplica armazenada no DN origem. Concluída a movimentação, a iteração é encerrada e as listas dos grupos são resetadas para as próximas iterações, que serão executadas caso o *cluster* – dado o *threshold* configurado – ainda não esteja balanceado.

4. Política de Balanceamento Customizada

Estudos passados atestaram que o desbalanceamento de réplicas afeta diretamente o desempenho do HDFS em atender aplicações voltadas a E/S [Fazul et al. 2019]. Motivado

por isso, definiu-se uma política de balanceamento customizada para o `HDFS Balancer` que baseia-se em prioridades – primariamente elencadas em [Fazul and Barcelos 2019] – dedicadas a flexibilizar e otimizar a operação do balanceador do Hadoop.

Para permitir funcionalidades de priorização na formação de pares de nodos origem-destino durante o balanceamento, a política customizada define um sistema de prioridades com base na topologia do *cluster* e em diferentes métricas do HDFS. As prioridades implementadas, separadas em categorias de acordo com suas características de funcionamento, são exibidas na Tabela 1. Todas as prioridades garantem que a disposição dos blocos após o balanceamento permaneça respeitando a PPR e que a variação máxima do volume de dados em cada DN continue sendo controlada pelo valor de *threshold*. Este trabalho é voltado às prioridades de "capacidade dos nodos", detalhadas na Seção 4.1.

Tabela 1. Sistema de prioridades definido pela política customizada.

Categoria	Prioridade	Objetivo geral
Capacidade dos nodos	Capacidade de armazenamento Capacidade de processamento	Customizar o balanceamento em ambientes heterogêneos a partir de diferenças de *hardware* dos DNs.
Estados dos nodos	Utilização dos nodos Classificação dos nodos Carga dos nodos	Reduzir o impacto da operação do balanceador nas demais aplicações rodando no HDFS através de priorizações com base em métricas recuperadas em tempo de execução.
Estados dos *racks*	Confiabilidade dos *racks* Utilização dos *racks*	Permitir que o balanceamento seja conduzido considerando características dos *racks* que agrupam os DNs do *cluster*.
Distribuição dos dados	Disponibilidade dos dados	Priorizar movimentações de réplicas que permitam aumentar a disponibilidade final dos blocos armazenados.

4.1. Priorização baseada na Capacidade dos Nodos

As prioridades de capacidade são dedicadas a instâncias do Hadoop executando em ambientes heterogêneos. De acordo com as diferenças de *hardware* das máquinas que rodam cada DN, a estratégia de reposicionamento das réplicas do `HDFS Balancer` é modificada. Os critérios para a priorização podem basear-se nas capacidades de processamento ou de armazenamento dos nodos. Ambas são similares em quesitos de implementação, diferindo apenas nas métricas utilizadas para determinar os nodos suscetíveis ao recebimento de uma maior ou menor quantidade de dados. A prioridade de **capacidade de processamento** baseia-se em diferenças na capacidade de computação dos DNs (quantidade de memória RAM). Já a prioridade de **capacidade de armazenamento** leva em consideração diferenças na capacidade de armazenamento dos DNs (total, utilizada e remanescente). Neste trabalho, o termo "capacidade", quando não especificado, varia de acordo com a prioridade configurada na etapa de balanceamento.

Para incorporar as priorizações, duas diretrizes devem ser seguidas: (i) um nodo com baixa capacidade deve ter prioridade em transferir dados; e (ii) um nodo com alta capacidade deve ter prioridade em receber dados. A primeira modificação a ser realizada no código-fonte do balanceador diz respeito à ordenação das listas responsáveis por manter os grupos de dispositivos de armazenamento dos nodos. Essas listas são separadas a partir da classificação dos $G_{i,t}$ realizada pelo balanceador. De modo a alcançar a diretriz (i), as listas dos grupos origem na transferência dos blocos (superutilizados e acima da média) devem ser ordenadas de forma decrescente baseadas na capacidade dos nodos (seja de processamento ou de armazenamento). Já para atingir a diretriz (ii), as listas dos grupos destino na redistribuição dos blocos (subutilizados e abaixo da média) devem ser ordenadas de forma crescente, também baseadas na capacidade dos nodos.

A segunda atualização está relacionada ao cálculo do volume de dados máximo permitido que pode ser movimentado do $G_{i,t}$ origem para o $G_{i,t}$ destino em cada iteração de balanceamento. Esse valor é determinado pela variável *maxSize2Move*. Com a política padrão do `HDFS Balancer`, *maxSize2Move* – cujo cálculo foi apresentado na Figura 1 da Seção 3.1.1 – representa a quantidade de dados necessária para levar a utilização de um determinado grupo de dispositivos ($U_{i,t}$) até a utilização média dos dispositivos de armazenamento tipo t do *cluster* ($U_{\mu,t}$). Com as modificações realizadas, a variável *maxSize2Move* passa a ser definida seguindo as diretrizes (i) e (ii).

Para tal, foi implementado um método dedicado à quantificação das capacidades de cada $G_{i,t}$ de acordo com a prioridade configurada. A Figura 2 apresenta o algoritmo definido no método *computeWeight*. Como somente é preciso executá-lo uma vez a cada iteração de balanceamento, a invocação do novo método foi incorporada em um momento anterior à chamada do método (padrão do balanceador) *Balancer.init*.

Figura 2. Método responsável por quantificar a capacidade de cada grupo.

```
1:  procedure COMPUTEWEIGHT(reports)
2:    for each r ∈ reports do
3:      for each t ∈ StorageTypes ∈ r do
4:        if t ⊄ capacityMap then
5:          capacitytMap.put(t, [∅])
6:        end if
7:        capacityMap.get(t).add(getPriorityCapacity(r, t))
8:      end for
9:    end for
10:   for each r ∈ reports do
11:     for each t ∈ StorageTypes ∈ r do
12:       min ← min(capacityMap.get(t))
13:       max ← max(capacityMap.get(t))
14:       weight ← 0.5
15:       if (max − min) ≠ 0 then
16:         weight ← (getPriorityCapacity(r, t) − min) / (max − min)        ▷ min-max (0, 1)
17:       end if
18:       key ← r.getDatanodeInfo().getDatanodeUuid() + ":" + t
19:       weightMap.put(key, weight)
20:     end for
21:   end for
22: end procedure
```

Na primeira parte do método *computeWeight* (L. 2 a 9), é preenchida a estrutura *capacityMap*, que relaciona cada tipo de dispositivo t com as capacidades totais dos grupos daquele tipo (a acumulação é realizada pelo método *add* na linha 7). O método auxiliar *getPriorityCapacity* retorna um valor que representa a capacidade do grupo de acordo com a prioridade configurada ($C_{i,t}$). Para a prioridade de **capacidade de armazenamento**, esse método irá realizar uma chamada ao método *getCapacity* (já utilizado pelo balanceador padrão), que devolve o valor em *bytes* referente a capacidade de armazenamento do $G_{i,t}$. Já para a prioridade de **capacidade de processamento**, o valor retornado equivale à capacidade de memória RAM total do nodo. Para estender a política customizada de modo a oferecer suporte a novas prioridades de capacidade, além de armazenamento e processamento (e.g. prioridade que considere características dos processadores de cada nodo), bastaria adicionar o retorno adequado em *getPriorityCapacity*.

Na segunda parte do método (L. 10 a 21), é criada a estrutura *weightMap*, que registra, para cada $G_{i,t}$, um valor que representa sua capacidade (C'_i). Para tal, aplica-se a normalização *min-max* em função da capacidade original de cada grupo ($C_{i,t}$), de modo que $C'_i = (C_{i,t} - min) \ / \ (max - min)$. A normalização *min-max* torna o valor mínimo de um conjunto de valores em 0, o valor máximo em 1 e, qualquer outro valor, em um decimal entre 0 e 1. Caso os $G_{i,t}$ não apresentem diferenças de capacidade, C'_i terá um valor médio de $0{,}5$ (L. 14), fazendo com que a posterior definição de *maxSize2Move* seja similar ao cálculo atual adotado pela política padrão do balanceador ($|U_{i,t}$ - $U_{\mu,t}|$).

Após computar a C'_i de cada $G_{i,t}$ (L. 16) e definir uma chave que representa o grupo (L. 18), cria-se uma entrada na estrutura *weightMap* (L. 19). Esta estrutura é posteriormente acessada pelo método *calcMaxSize2MoveBasedOnNodeCapacity* (Figura 3), que foi implementado de forma a aumentar ou diminuir o valor de *maxSize2Move* – respeitando a propriedade *dfs.balancer.max-size-to-move* – com base na C'_i de cada grupo. Sua chamada foi incorporada ao método *Balancer.init*, de modo que quando o `HDFS Balancer` for executado com alguma prioridade, o cálculo padrão de *maxSize2Move* seja substituído pela variação implementada na política customizada. Ressalta-se que o parâmetro *capacity* desse método refere-se à capacidade de armazenamento do $G_{i,t}$ (não confundir com a capacidade normalizada – C'_i – armazenada no *weightMap*), sendo necessária para o cálculo independente da prioridade de capacidade configurada.

Figura 3. Método dedicado ao cálculo customizado da variável *maxSize2Move*.

```
 1: procedure CALCMAXSIZE2MOVEBASEDONNODECAPACITY(r, t, capacity, utilization, average)
 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:   key ← r.getDatanodeInfo().getDatanodeUuid() + ":" + t
 9:   if utilizationDiff > 0 then                                          ▷ origem
10:     if thresholdDiff ≤ 0 then                                  ▷ G_{i,t} acima da média
11:       weightBasedBytes ← (bytes2InfLim + bytes2SupLim) × (1 − weightMap.get(key))
12:       maxSize2Move ← max(0, weightBasedBytes − bytes2SupLim)
13:     else                                                       ▷ G_{i,t} superutilizado
14:       weightBasedBytes ← bytes2InfLim × (1 − weightMap.get(key))
15:       maxSize2Move ← max(bytes2SupLim, weightBasedBytes)
16:     end if
17:   else                                                                ▷ destino
18:     if thresholdDiff ≤ 0 then                                  ▷ G_{i,t} abaixo da média
19:       weightBasedBytes ← (bytes2InfLim + bytes2SupLim) × weightMap.get(key)
20:       maxSize2Move ← max(0, weightBasedBytes − bytes2InfLim)
21:     else                                                       ▷ G_{i,t} subutilizado
22:       weightBasedBytes ← bytes2SupLim × weightMap.get(key)
23:       maxSize2Move ← max(bytes2InfLim, weightBasedBytes)
24:     end if
25:     maxSize2Move ← min(getRemaining(r, t), maxSize2Move)
26:   end if
27:   return min(maxSize2Move, max)                                  ▷ max = 10GB
28: end procedure
```

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 C_i' do grupo. De forma geral, o valor da variável *maxSize2Move* para um $G_{i,t}$ com uma C_i' elevada aproxima-se do volume de *bytes* necessário para elevar sua $U_{i,t}$ até o limite superior. Já para um $G_{i,t}$ com uma C_i' baixa, aproxima-se do volume de *bytes* necessário para reduzir sua $U_{i,t}$ até o limite inferior. Para idealizar este comportamento, é importante observar alguns detalhes. Em um grupo origem, *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, *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 C_i' 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. 12), garantindo assim que sua utilização não seja reduzida. Similarmente, em um $G_{i,t}$ abaixo da média, se sua C_i' resultar em uma utilização inferior à $U_{i,t}$ atual do grupo, *maxSize2Move* também deverá ter o valor zero (L. 20), garantindo que sua utilização não seja aumentada. Além disso, em grupos superutilizados com alta C_i', 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 15). Já em grupos subutilizados com baixa C_i', o valor de *maxSize2Move* deve, ao mínimo, ser o suficiente para permitir a classificação do grupo como abaixo da média (garantido pelo método *max* na linha 23).

Por fim, são aplicadas duas validações (definidas também no cálculo realizado pela política padrão): (i) em grupos destino, *maxSize2Move* deve ser inferior ao espaço de armazenamento remanescente no $G_{i,t}$ (L. 25); e (ii) *maxSize2Move* deve respeitar o valor de *max* (dado pela propriedade *dfs.balancer.max-size-to-move*) (L. 27).

5. Experimentação

Os experimentos foram realizados na plataforma GRID'5000[2] com o Hadoop (versão 2.9.2) em modo de operação totalmente distribuído. Tendo em vista que as prioridades de capacidade são voltadas a ambientes heterogêneos, o HDFS foi configurado com múltiplos *racks* contendo nodos com diferenças de *hardware* entre si.

Ao total, foram configurados 24 nodos dispostos em 3 *racks* lógicos distintos (R_1, R_2 e R_3) do *site Rennes*, onde cada *rack* executou, respectivamente, 8, 6 e 10 DNs e cada DN mantinha um único dispositivo de armazenamento do tipo *DISK*. Os DNs do *rack* R_1 pertenciam ao *cluster paravance* e cada um possuía 2 processadores Intel Xeon E5-2630 v3 (8 cores por CPU), 128GB de memória RAM e 509,68GB de capacidade de armazenamento, com duas conexões *Ethernet* de 10Gbps. Os DNs do *rack* R_2, pertencentes ao *cluster parapide*, possuíam 2 processadores Intel Xeon X5570 (4 cores por CPU), 24GB de memória RAM e 417,99GB de capacidade de armazenamento, com uma conexão *InfiniBand* de 20Gbps. Já os DNs do *rack* R_3, pertencentes ao *cluster parapluie*, possuíam

[2]Grid'5000 é uma plataforma para experimentos apoiada por um grupo de interesses científicos hospedado pelo Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais detalhes em `https://www.grid5000.fr`).

2 processadores AMD Opteron 6164 HE (12 cores por CPU), 48GB de memória RAM, 188,77GB de capacidade de armazenamento e conexão *InfiniBand* de 20Gbps.

Devido às configurações do ambiente de testes, a prioridade de balanceamento avaliada nesta seção refere-se à **capacidade de armazenamento** dos nodos. Assim, a normalização *min-max* foi aplicada com base no espaço de disco total dos DNs ($C_{i,DISK}$). As capacidades normalizadas (C'_i) para os DNs dos *racks* R_1, R_2 e R_3 foram de, respectivamente, $1{,}0$, $0{,}7$ e $0{,}0$. Para avaliar a distribuição dos blocos, 10 arquivos de 40GB com um FR de 3 réplicas por bloco foram escritos no HDFS através do *benchmark* `TestDFSIO` [White 2015], totalizando um volume de dados de 1,2TB.

5.1. Resultados e Discussão

Ao total, o HDFS possuía um espaço de armazenamento de 8,27TB estando com 1,22TB ocupados ($U_{\mu,DISK}$ em $14{,}75\%$). Inicialmente, o *rack* R_1 mantinha 828,93GB dos dados (utilização em $20{,}33\%$), o *rack* R_2 mantinha 289,46GB (utilização em $11{,}54\%$) e o *rack* R_3 mantinha 129,5GB (utilização em $6{,}86\%$). A Tabela 2 apresenta a ocupação ($O_{i,t}$) e a porcentagem de utilização ($U_{i,t}$) de cada DN sem balanceamento de réplicas (distribuição dos dados inicial, baseada na PPR) e após a execução do `HDFS Balancer` configurado com a prioridade de capacidade de armazenamento e um *threshold* de 5%.

Tabela 2. Estado do HDFS antes e após o balanceamento de réplicas com a prioridade de capacidade de armazenamento dos nodos.

Rack	$C_{i,DISK}$	C'_i	DataNode	s/ balanceamento		c/ balanceamento	
				$O_{i,DISK}$ (GB)	$U_{i,DISK}$ (%)	$O_{i,DISK}$ (GB)	$U_{i,DISK}$ (%)
R_1	509,68	1,0	DN_{01}	137,56	26,99	96,89	19,01
			DN_{02}	127,47	25,01	96,89	19,01
			DN_{03}	109,43	21,47	96,89	19,01
			DN_{04}	108,51	21,29	96,89	19,01
			DN_{05}	111,11	21,80	96,89	19,01
			DN_{06}	81,85	16,06	81,85	16,06
			DN_{07}	76,50	15,01	76,50	15,01
			DN_{08}	76,50	15,01	76,50	15,01
R_2	417,99	0,7	DN_{09}	83,26	19,92	62,11	14,86
			DN_{10}	0,00	0,00	46,10	11,03
			DN_{11}	97,77	23,39	62,24	14,89
			DN_{12}	54,17	12,96	55,63	13,31
			DN_{13}	54,26	12,98	55,38	13,25
			DN_{14}	0,00	0,00	45,35	10,85
R_3	188,77	0,0	DN_{15}	21,26	11,26	21,26	11,26
			DN_{16}	0,00	0,00	18,67	9,89
			DN_{17}	44,04	23,33	27,99	14,83
			DN_{18}	0,00	0,00	18,67	9,89
			DN_{19}	0,00	0,00	18,71	9,91
			DN_{20}	23,12	12,25	23,12	12,25
			DN_{21}	13,61	7,21	17,82	9,44
			DN_{22}	13,86	7,34	18,74	9,93
			DN_{23}	13,61	7,21	18,74	9,93
			DN_{24}	0,00	0,00	18,86	9,99

O *rack* R_1 possuía 5 nodos superutilizados (DN_{01} a DN_{05}), i.e. com utilização acima de $19{,}75\%$ ($U_{\mu,DISK}$ + *threshold*). Tendo em vista a capacidade normalizada (C'_i) dos nodos do *rack* em questão ($1{,}0$), a utilização dos DNs superutilizados após o balanceamento ficou próxima ao limite superior. Já os demais DNs desse *rack* (DN_{06} a DN_{08}) estavam classificados como acima da média. Dessa forma, conforme citado durante o relato da implementação dessa prioridade, a utilização desses nodos não poderia ser estendida

para além da atual. Sendo assim, dada a C'_i em $1{,}0$, o volume de dados armazenado nos DNs com utilização acima da média foi mantido inalterado.

Em relação ao *rack* R_2, haviam 2 nodos subutilizados (DN_{10} e DN_{14}), com utilização abaixo de $9{,}75\%$ ($U_{\mu,DISK} - \textit{threshold}$), 3 nodos classificados como abaixo da média (DN_{09}, DN_{12} e DN_{13}) e 1 único nodo superutilizado (DN_{11}). Dada a C'_i dos nodos desse *rack* $(0{,}7)$, suas utilizações foram levadas, na medida do possível, para próximo da $U_{\mu,DISK}$. Como nem sempre há um volume de dados suficiente a ser transferido por outros nodos do *cluster*, a utilização de determinados DNs desse *rack* (DN_{10} e DN_{14}) ficou abaixo do que seria esperado em função de suas capacidades normalizadas.

Os DNs do *rack* R_3, por sua vez, possuíam a menor C'_i do *cluster* $(0{,}0)$. Assim, a prioridade de capacidade esforça-se em reduzir ao máximo suas utilizações. Após o balanceamento, todos os nodos subutilizados desse *rack* ficaram com utilização próxima ao limite inferior $(9{,}75\%)$. Já os DNs classificados como abaixo da média (DN_{15} e DN_{20}) e com C'_i em $0{,}0$, não poderiam ter sua utilização reduzida para além da atual e, dessa forma, o volume de dados nestes nodos foi mantido inalterado. De forma similar, o DN_{17} (superutilizado) ficou com utilização próxima a $U_{\mu,DISK}$ (dado que, no momento que sua $U_{i,t}$ ficasse abaixo da $U_{\mu,DISK}$, ele estaria na mesma situação que o DN_{15} e o DN_{20}).

Tendo as capacidades normalizadas dos nodos com base no agrupamento por *rack*, onde $C'_{R_1} > C'_{R_2} > C'_{R_3}$, observa-se que o *rack* cujos nodos possuíam a maior capacidade calculada (R_1) ficou responsável por manter um maior volume de dados (719,3GB ao total e utilização em $17{,}64\%$) que o *rack* com a segunda maior capacidade (R_2) (326,81GB e utilização em $13{,}03\%$). Enquanto isso, o *rack* com menor capacidade (R_3), ficou com a menor ocupação dentre os demais (162,65GB e utilização de $10{,}78\%$), validando assim o objetivo da prioridade de capacidade de armazenamento dos nodos.

De forma a avaliar o impacto do balanceamento com a prioridade de capacidade de armazenamento no desempenho do sistema, mediu-se o tempo necessário para a leitura dos dados com o *benchmark* `TestDFSIO`. A Figura 4 exibe os tempos de 20 execuções distintas do *benchmark*. Inicialmente, o tempo médio de leitura foi de 950,18 segundos. Com o balanceamento de réplicas, esse valor foi reduzido para 806,83 segundos. 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 de execução nas 20 execuções do *benchmark* antes e após a execução do balanceador, a variação alcançada foi de $-15{,}09\%$, indicando a redução no tempo necessário para a leitura dos dados.

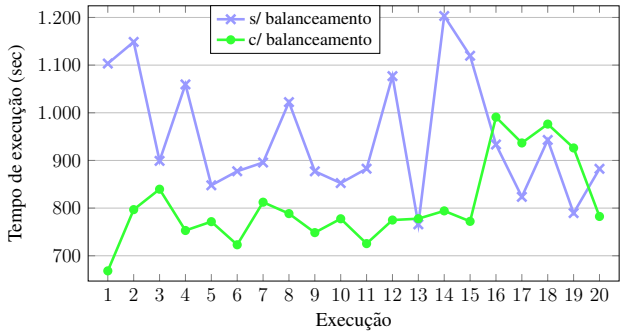

Figura 4. Tempo para leitura dos dados antes e após o balanceamento com a prioridade de capacidade de armazenamento dos nodos.

6. Considerações Finais

Este trabalho apresentou uma estratégia de balanceamento de réplicas customizada para o `HDFS Balancer` baseada em prioridades de capacidade, que são otimizadas para instâncias do Hadoop executando em ambientes heterogêneos. A solução proposta visa flexibilizar a operação do balanceador nativo do HDFS de forma que diferenças nas capacidades de armazenamento ou de processamento sejam utilizadas como forma de priorização dos nodos para o recebimento de um maior ou menor volume de dados durante o balanceamento do *cluster*. Após detalhar o funcionamento das prioridades, foram conduzidos experimentos voltados a validar e avaliar a efetividade de implementação.

Os resultados demostram que a priorização do balanceamento do HDFS em função das diferenças nas capacidades dos nodos permite uma exploração otimizada da localidade dos dados, aprimorando o desempenho geral do sistema de arquivos do Hadoop em atender operações de entrada e saída. Trabalhos futuros envolvem uma investigação aprofundada das prioridades em ambientes que permitam endereçar demandas de uso específicas de aplicações reais. Em complemento, pretende-se estender a implementação de modo a permitir que as prioridades de capacidade de armazenamento e de processamento dos nodos possam ser utilizadas de forma associada pelo `HDFS Balancer`.

Referências

Achari, S. (2015). *Hadoop Essentials*. Packt Publishing Ltd, Birmingham, 1st edition.

Fazul, R. W. A. and Barcelos, P. P. (2019). Política customizada de balanceamento de réplicas para o hdfs balancer do apache hadoop. In *Anais do XX Workshop de Testes e Tolerância a Falhas*, pages 93–106. SBC.

Fazul, R. W. A., Cardoso, P. V., and Barcelos, P. P. (2019). Análise do impacto da replicação de dados implementada pelo apache hadoop no balanceamento de carga. In *Anais do X Computer on the Beach*, pages 579–588, Florianópolis. Univali.

Foundation, A. S. (2018). "HDFS Architecture". `hadoop.apache.org/docs/r2.9.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign`. Junho.

Hortonworks (2018). "HDFS Administration". `https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.5/bk_hdfs-administration/content/ch_balancing-in-hdfs.html`. Junho.

Ibrahim, I. A., Dai, W., and Bassiouni, M. (2016). Intelligent data placement mechanism for replicas distribution in cloud storage systems. In *IEEE International Conference on Smart Cloud (SmartCloud)*, pages 134–139, New York. IEEE.

Liu, K., Xu, G., and Yuan, J. (2013). An improved hadoop data load balancing algorithm. *Journal of Networks*, 8(12):2816–2822.

Shah, A. and Padole, M. (2018). Load balancing through block rearrangement policy for hadoop heterogeneous cluster. In *2018 Int. Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 230–236, Bangalore. IEEE.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE.

White, T. (2015). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4 edition.