UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

RENAN IGLESIAS ALVES DE REZENDE

UM ESTUDO DE COMPRESSÃO DE GRANDES VOLUMES DE DADOS USANDO O *FRAMEWORK* HADOOP

Rio de Janeiro
2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

RENAN IGLESIAS ALVES DE REZENDE

UM ESTUDO DE COMPRESSÃO DE GRANDES VOLUMES DE DADOS USANDO O *FRAMEWORK* HADOOP

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Silvana Rossetto

Rio de Janeiro
2015

CBIB de Rezende, Renan Iglesias Alves

Um estudo de compressão de grandes volumes de dados usando o *framework* Hadoop / Renan Iglesias Alves de Rezende. – 2015.
114 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática, Rio de Janeiro, 2015.

Orientador: Silvana Rossetto.
.

– Teses. I. Rossetto, Silvana (Orient.). II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em Informática. III. Título

CDD

RENAN IGLESIAS ALVES DE REZENDE

Um estudo de compressão de grandes volumes de dados usando o *framework* Hadoop

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de _____ de _____.

Prof[a]. Dr[a]. Silvana Rossetto (Orientador)

Prof[a]. Dr[a]. Flávia Coimbra Delicato

Prof. Dr. Gabriel Pereira da Silva

Prof. Dr. Eugene Francis Vinod Rebello

A todos aqueles que me apoiaram e acreditaram em mim durante esta caminhada.

AGRADECIMENTOS

Foi uma caminhada bem longa (mais do que eu gostaria), bem árdua, um grande desafio que enfrentei durante todo este tempo. Sem dúvida nenhuma, muitas pessoas contribuíram, direta ou indiretamente, para que eu conseguisse chegar ao final deste trabalho. Deixo meus sinceros agradecimentos a todos vocês que fizeram parte desta minha caminhada.

Em primeiro lugar, agradeço à toda minha família pelo apoio incondicional e pelo carinho, que me deram forças para continuar, mesmo nos momentos que eu tive vontade de jogar tudo pro alto e por me mostrar o caminho certo a ser seguido nesta vida.

Faço também, um agradecimento a todos os meus professores desta caminhada e também das anteriores, onde tive a oportunidade de aprender um pouco com cada um deles. Em especial, agradeço à minha orientadora, Prof^{a}. Silvana Rossetto, que me acolheu em um momento de muitas dúvidas e foi uma pessoa com a qual tive a grande oportunidade de trabalhar durante este projeto e adquirir um grande aprendizado. Todas suas orientações e seus auxílios foram essenciais para a confecção deste trabalho, assim como sua compreensão nos momentos em que nada dava certo (momentos não muito raros).

Agradeço a todos os meus amigos e conhecidos, principalmente os que compreenderam meus momentos de sumiço da vida social para poder focar neste trabalho. Em especial, deixo registrado um grande "muito obrigado"ao Cláudio Marassi, e ao casal mais amado do Brasil, Renato Lond e Nat Veiga, que não se importaram de dedicar horas preciosas de suas vidas para me ajudar nos momentos de dificuldade, inclusive finais de semana. E tudo isso sem exigir nada em troca! Agradeço também a todos os outros que se ofereceram caso eu precisasse de ajuda, mas que eu preferi não incomodar!

Renan Iglesias Alves de Rezende

RESUMO

de Rezende, Renan Iglesias Alves. **Um estudo de compressão de grandes volumes de dados usando o *framework* Hadoop**. 2015. 103 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

A cada ano observa-se um aumento significativo da quantidade de dados gerados por diferentes aplicações. O mapeamento geológico de uma determinada região, por exemplo, pode implicar na necessidade de armazenar vários arquivos de dados na ordem de giga ou terabytes. Técnicas de compressão de dados são normalmente usadas para reduzir o espaço de memória ocupado por esses dados e diminuir seu custo de manutenção. Entretanto, comprimir arquivos na ordem de terabytes pode demandar um tempo de processamento significativo quando usamos algoritmos que executam em uma única máquina devido principalmente à sobrecarga de operações de entrada e saída e à restrição de espaço de memória para processamento dos dados. Neste trabalho, propomos uma arquitetura distribuída e flexível para executar algoritmos de compressão de dados de forma paralela, usando o ambiente Hadoop YARN. A característica de flexibilidade visa permitir que as aplicações se adaptem à infraestrutura de hardware oferecida, utilizando-a de forma eficiente e escalável, em um ambiente que pode ser compartilhado com outras aplicações. Utilizaremos o framework Hadoop YARN de duas formas: (i) usando o modelo de programação MapReduce; e (ii) desenvolvendo um modelo particular de programação mais adequado para os algoritmos de compressão de dados, executado diretamente sobre o Hadoop YARN. Usaremos como referência o algoritmo de compressão Huffman. Trata-se de um algoritmo de compressão sem perdas bastante conhecido e que apresenta características básicas de vários algoritmos de compressão, entre as quais a necessidade de realizar mais de uma leitura do arquivo de entrada.

Palavras-chave: .

ABSTRACT

de Rezende, Renan Iglesias Alves. **Um estudo de compressão de grandes volumes de dados usando o *framework* Hadoop**. 2015. 103 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

In the later years, we can observe a significant increase in the amount of data generated by different applications. The geological mapping of a given region, for example, may generate multiple data files in the order of giga or terabytes. Data compression techniques are normally used to reduce the memory space occupied by files and then minimize its maintenance cost. However, compress files in the order of gigabytes or terabytes may require significant processing time when executing algorithms on a restrict scope of a single machine mainly due to the overload of input and output operations and the restriction of memory space for data processing. In this paper, we propose a distributed and scalable architecture to execute data compression algorithms by using the Hadoop YARN framework. The purpose is to ensure that the compression algorithms to adapt to the available hardware infrastructure, using it in an efficient and scalable way, in an environment that can be shared with other applications. We will use the framework Hadoop YARN in two ways: (i) using the MapReduce programming model; and (ii) developing a particular programming model most suitable for data compression algorithms, performed directly on Hadoop YARN. We will use as a reference the Huffman coding. It is a well know lossless compression algorithm with basic characteristics of various compression algorithms, including the need for more than one read from the input file.

Keywords: Distributed data compression, Hadoop YARN framework, MapReduce programming model, Huffman coding.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, a área da computação obteve grandes avanços tecnológicos que resultaram no crescimento da quantidade de dados armazenados e processados. Como exemplo disto, podem-se citar: aplicações de redes de sensores onde os nós captam grandes quantidades de dados relativos ao ambiente onde se encontram; sistemas utilizados por grandes empresas de venda onde são atendidas milhares de requisições de clientes ao redor do mundo [16]; sistemas computacionais com algoritmos complexos de análise numérica para resolver diversos tipos de problemas do mundo real, como aplicações de análise e processamento de dados sísmicos ou genéticos, onde o tamanho dos arquivos de entrada são da ordem de gigabytes ou terabytes [1]. O crescimento da quantidade de dados digitais gerados está diretamente ligado à necessidade de soluções de armazenamento. Um exemplo comum é a rotina existente em muitas empresas de salvar cópias de todos os seus dados sob a forma de backup. Todos os casos citados podem gerar ou utilizar dados massivos e necessitam acessá-los de forma eficiente e em tempo viável.

A compressão de dados surge, nesse contexto, como uma alternativa, pois é capaz de diminuir os custos financeiros e operacionais (na medida em que discos rígidos são extremamente sensíveis e suscetíveis a falhas), além de permitir, com a diminuição do tamanho dos arquivos, a redução da sobrecarga da rede e do tempo necessário para a transmissão de arquivos de um lugar a outro [7]. Muitos algoritmos já existem para realizar a compressão de dados e eles podem ser divididos em duas principais categorias, de acordo com a ocorrência ou não de perda de dados do arquivo original durante os processos de compressão e descompressão. A primeira delas é a dos algoritmos que realizam compressão retirando a redundância de dados de um arquivo, baseados no fato de que alguns dados podem ser eliminados e de-

pois reconstruídos de acordo com alguma aproximação. Este tipo de compressão é realizado pelos algoritmos de compressão com perda, pois um arquivo, após passar pelos processos de compressão e descompressão, não é exatamente igual ao original. A segunda categoria é a dos algoritmos de compressão sem perda, onde os arquivos são comprimidos de forma que cada símbolo do arquivo original seja codificado e corresponda a outro símbolo de menor tamanho. A relação entre os símbolos do arquivo original e do arquivo comprimido geralmente é armazenada em um cabeçalho no próprio arquivo comprimido que, ao ser descomprimido, é idêntico ao arquivo original [25].

Os algoritmos de compressão sem perda são utilizados para comprimir arquivos onde a alteração de um único dado compromete sua integridade, tornando-os inutilizáveis. Em um arquivo executável, por exemplo, deve-se utilizar esse tipo de compressão, pois, caso seja utilizada a compressão com perda pode ocorrer a troca de uma instrução por outra, ou a perda de alguma instrução no momento da reconstrução do arquivo descompactado, comprometendo seu funcionamento. Analogamente, a troca ou perda de um simples caracter em um arquivo de texto pode torná-lo ilegível ou alterar o assunto nele contido. Já os algoritmos de compressão com perda são usualmente utilizados para comprimir arquivos de imagem ou som, nos quais uma pequena alteração em alguns dos dados pode não ser perceptível para o usuário, como alguns pixels em uma imagem, que terá suas cores levemente alteradas. Cada método tem suas vantagens e desvantagens, como tempo de processamento, taxa de compressão, dentre outros, porém a decisão sobre qual método deverá ser utilizado fica restringida pelo tipo de arquivo a ser comprimido.

1.1 Motivação

Um dos grandes problemas dos algoritmos de compressão, em particular dos algoritmos sem perda de dados, é o tempo de execução, devido, em grande parte, à necessidade de leitura e escrita dos dados em uma memória persistente (normalmente discos rígidos). No caso da compressão de dados massivos, o tempo total de processamento pode tornar-se significativamente demorado. Dependendo da aplicação, é necessário que essa compressão seja feita de forma mais rápida, então, melhorias de software e/ou hardware tornam-se essenciais.

O hardware dos computadores sofreu avanços consideráveis relacionados ao seu poder de processamento nos últimos anos, como o surgimento das arquiteturas multiprocessadores que permitem a execução de várias tarefas de uma mesma aplicação simultaneamente em uma única máquina. Outro avanço significativo ocorreu em relação ao hardware das GPUs (*Graphics Processor Unit*), que passou a ser uma alternativa para a computação de alto desempenho. Com relação ao armazenamento dos dados, que geralmente é feito em uma memória persistente, os avanços em termos de velocidade de acesso não foram tão significativos. Como este tipo de armazenamento geralmente envolve dispositivos mecânicos, os mesmos acabam tornando-se fator limitante no aumento da velocidade de armazenamento. Algumas alternativas foram criadas permitindo-se melhorar o desempenho no acesso aos dispositivos de armazenamentos não-voláteis, como o RAID (*Redundant Array of Independent Drives*) e os sistemas de arquivos paralelos. Estas soluções, porém, são custosas financeiramente e operacionalmente, o que não permitiu que elas fossem utilizadas em larga escala em ambientes domésticos.

Na computação, uma aplicação pode ter seu desempenho limitado por dois fatores principais: poder de processamento e tempo para realizar operações de I/O. As aplicações *CPU bound* têm a característica de serem limitadas pelo poder de

processamento da máquina onde são executadas. Como exemplo, podemos citar uma sequência de cálculos matriciais, onde a maior parte do tempo gasto por ela é durante a execução de uma grande quantidade de cálculos. Já as aplicações chamadas *I/O bound* são as que executam muitas operações de entrada e/ou saída de dados. Uma aplicação que faz a cópia de um arquivo, por exemplo, é classificada como *I/O bound*.

Todos os exemplos de soluções criados para melhorar o desempenho das aplicações podem funcionar muito bem em alguns casos, porém podem não ter desempenho satisfatório em outros. Para a execução de algumas aplicações, as arquiteturas *multiprocessadores* podem ser suficientes, porém pode ser necessário que a capacidade de processamento seja ainda maior. Nestes casos, o uso combinado de CPUs e GPUs e de arquiteturas de processamento distribuído tornam-se uma alternativa. Porém, para aplicações *I/O bound* nem sempre essas alternativas são suficientes para que a aplicação obtenha um tempo de execução satisfatório devido ao gargalo das operações de I/O. Nestes casos, além do uso de processamento distribuído, é necessária a adoção de um sistema de arquivos paralelo, que, diferentemente dos sistemas de arquivos distribuídos, tem por objetivo otimizar acessos paralelos aos dados.

A programação de aplicações paralelas e distribuídas é uma tarefa complexa, uma vez que cabe ao programador gerenciar a distribuição balanceada das tarefas entre os nós e a interação entre eles, e além disso lidar com as possibilidades de falhas de comunicação ou específicas de cada nó. Diversas bibliotecas, APIs (*Application Programming Interface*) e modelos foram criados para auxiliar na implementação de aplicações paralelas e distribuídas em diferentes linguagens de programação. Para que seja possível utilizar um sistema distribuído, é necessário configurar um ambiente composto por um conjunto de hardware e software. A parte de hardware inclui os computadores (ou nós) e uma rede de interconexão, enquanto a parte de software inclui algoritmos para gerenciar os nós e distribuir as tarefas entre eles. Para permitir a interação entre os nós, é necessário dispor de uma interface de troca de mensagens

ou de uma arquitetura virtual de memória compartilhada. Ao usar uma biblioteca que siga o modelo de troca de mensagens, como o MPI (*Message Passing Interface*), os nós se comunicam uns com os outros enviando mensagens através de sua rede de interconexão. Um simples exemplo é uma aplicação que realiza a soma de todos os números inteiros armazenados por um vetor. Neste modelo, cada nó realiza a soma da parte do vetor pela qual é responsável e envia este valor, via mensagem de rede, a um nó que realizará a soma total. Já em uma aplicação que use uma biblioteca para implementar o modelo de memória compartilhada, um espaço de memória virtual é criado e compartilhada por todos os nós. Quando um nó tenta acessar uma determinada variável da aplicação, esta variável pode estar fisicamente armazenada em um outro nó, mas é acessível logicamente por todos, através da rede de interconexão. Condições de sincronização e exclusão mútua podem ser necessárias neste modelo. A ideia principal destas bibliotecas e APIs é criar uma abstração para o programador, fazendo com que ele não precise se preocupar com algumas destas tarefas complicadas de comunicação entre os nós, por exemplo.

Nos últimos anos, o *framework* Hadoop, juntamente com o modelo de programação MapReduce, popularizou-se neste cenário como alternativa para facilitar o desenvolvimento de aplicações paralelas e distribuídas. O Hadoop é um *framework* que possibilita o processamento paralelo e distribuído de grandes quantidades de dados em clusters de computadores utilizando modelos de programação mais simples, diminuindo-se, assim, o trabalho necessário para o desenvolvimento de aplicações paralelas. Ele foi projetado para oferecer alta escalabilidade, possibilitando o seu uso em clusters com poucos nós ou com milhares de nós, onde cada um oferece serviços de processamento e armazenamento de dados [23]. Ao invés de tentar suprir a necessidade de alta disponibilidade dos serviços aumentando a qualidade do hardware, este *framework* possui uma biblioteca criada para detectar e gerenciar falhas, fazendo com que o cluster disponibilize às suas aplicações um sistema de alta disponibilidade, mesmo quando ele é composto de máquinas de baixo custo suscetíveis a

falhas. O Hadoop possui um sistema de arquivos distribuído próprio que é útil para o armazenamento e acesso eficiente de arquivos de grandes tamanhos.

As facilidades oferecidas pelo Hadoop, como tolerância a falhas e modelos de programação mais simples para os desenvolvedores de aplicações paralelas, foram fatores que contribuíram para o sucesso deste *framework*. Pequenas e grandes empresas ao redor do mundo estão utilizando o Hadoop para o armazenamento e processamento de *Big Data* de forma simples e eficiente [12]. Aliado à utilização em larga escala deste *framework*, outro fator que evidencia seu sucesso é o fato de vários serviços de *cloud computing* oferecerem *templates* já prontos para que seus usuários montem facilmente um ambiente na nuvem que os possibilite executar suas aplicações de forma paralela e distribuída usando o Hadoop. Entre os serviços mais conhecidos, podem-se citar o Microsoft Azure [5] e o AWS, da Amazon [4].

Apesar do seu grande sucesso, o modelo de progamação MapReduce nem sempre se adequa às características das aplicações, como o processamento de grafos que não pode ser escrito segundo este modelo. Em função disso, a partir da versão 2.0, o Hadoop desvinculou seus componentes principais de gerência de execução e tolerância a falhas do modelo de programação MapReduce, apresentando sua nova versão chamada Hadoop YARN [28]. O objetivo foi facilitar a construção e utilização de outros modelos de programação distribuída sobre a mesma infraestrutura de gerência dos recursos de hardware, permitindo, de forma transparente, o compartilhamento de um cluster de máquinas por diferentes aplicações usando modelos de programação distintos.

1.2 Objetivos do Trabalho

Neste trabalho, propomos uma implementação distribuída e flexível para executar algoritmos de compressão de dados de forma paralela, usando o ambiente Hadoop YARN em clusters computacionais. A característica de flexibilidade visa permitir que as aplicações se adaptem à infraestrutura de hardware oferecida pelo cluster, utilizando os recursos disponíveis (CPU e memória) de forma eficiente e escalável, em um ambiente que pode ser compartilhado com outras aplicações. Utilizaremos o *framework* Hadoop YARN de duas formas: (i) adotando o modelo de programação MapReduce (componente das versões 1.0 do Hadoop); e (ii) desenvolvendo um modelo particular de programação mais adequado para os algoritmos de compressão de dados executado diretamente sobre o Hadoop YARN, que permite que aplicações que não seguem o modelo MapReduce sejam executadas sobre clusters Hadoop.

Usaremos o algoritmo de compressão Huffman como referência para o nosso trabalho. Trata-se de um algoritmo de compressão sem perdas, classificado na categoria de métodos de compressão estatísticos [20]. A opção por esse algoritmo deve-se ao fato de se tratar de um algoritmo bastante conhecido e que apresenta características básicas de vários algoritmos de compressão, entre as quais a necessidade de realizar mais de uma leitura do arquivo de entrada. Uma destas leituras envolve medidas estatísticas do arquivo de entrada, o que apresenta um ótimo desempenho quando realizado sobre o *framework* Hadoop. A codificação de Huffman é usada como método único, ou como um dos métodos de compressão em cascata, de vários algoritmos de compressão, como por exemplo, GZIP, PKZIP e BZIP2.

1.3 Contribuições

Como contribuição, este trabalho realiza um estudo sobre alternativas de paralelização de algoritmos de compressão de dados, utilizando as facilidades providas pelo *framework* Hadoop. Diferentes estratégias de paralelização — utilizando-se o algoritmo de Huffman como referência — foram implementadas e avaliadas, visando explorar ao máximo os recursos oferecidos pelo *framework* Hadoop.

Acreditamos que as estratégias avaliadas podem ser generalizadas e utilizados como base para a criação de um modelo de programação específico para compressão de dados que executa no ambiente Hadoop, facilitando o desenvolvimento de aplicações de compressão de dados paralelas que façam uso de outros algoritmos ou técnicas de compressão.

1.4 Organização do Texto

O restante deste texto está organizado da seguinte forma. No Capítulo 2 apresentamos os conceitos básicos sobre o problema de compressão de dados e como funciona o algoritmo de Huffman, e também sobre o framework Hadoop YARN e o modelo de programação MapReduce. No Capítulo 3 apresentamos uma discussão sobre trabalhos relacionados. No Capítulo 4 apresentamos as nossas propostas de implementações distribuídas para compressão de grandes volumes de dados usando o algoritmo de Huffman e o framework Hadoop. Também é apresentada uma discussão sobre as possibilidades de generalização dessas implementações para permitir o uso de diferentes algoritmos ou técnicas de compressão de dados. Após isso, no Capítulo 5 descrevemos a metodologia, as métricas e o ambiente de experimentação utilizado para avaliar a implementação proposta, além dos resultados obtidos e as

discussões sobre eles. Por fim, no Capítulo 6, apresentamos as discussões finais e conclusões deste trabalho.

2 CONCEITOS BÁSICOS

Neste capítulo, apresentaremos uma visão geral sobre o problema de compressão de dados e a codificação de Huffman. Em seguida, detalharemos as características gerais do framework para programação distribuída Hadoop YARN e sua implementação do modelo de programação MapReduce. Por fim, apresentaremos uma breve discussão sobre clusters computacionais e seu uso para programação paralela e distribuída.

2.1 Algoritmos de Compressão de Dados

Comprimir dados significa diminuir o espaço de armazenamento ocupado por eles. A compressão surgiu devido ao grande crescimento dos dados gerados e armazenados, associado ao custo de armazenamento (financeiro e operacional), e também à necessidade de redução de sobrecarga das redes por onde estes dados são transmitidos.

A compressão é realizada por algoritmos que possuem dois componentes principais: um codificador, compressor ou *encoder*, e um decodificador, descompressor ou *decoder*. Enquanto o primeiro é responsável pela compressão de um ou mais arquivos, o segundo é responsável pela descompressão deles. O fluxo de compressão e descompressão ocorre da seguinte forma: uma sequência de bytes, que representa um ou mais arquivos, é passada como entrada do compressor que, seguindo uma sequência de cálculos ou transformações especificadas pelo algoritmo, gera uma outra sequência de bytes, menor que a original, representando esta entrada comprimida. No momento de utilizar o arquivo comprimido, é necessário utilizar o descompressor,

que receberá como entrada o arquivo comprimido e, seguindo uma série de cálculos ou transformações de acordo com o algoritmo utilizado, gerará um arquivo igual ou similar ao original. O que determina se este arquivo, após comprimido e descomprimido, será igual ou similar ao arquivo original, é o algoritmo de compressão utilizado, que pode ser classificado como com perda de dados ou sem perda, conforme citado anteriormente.

Outra classificação para os algoritmos utilizados para compressão (tanto os com perda quanto os sem perda de dados) pode ser feita levando-se em consideração o método de operação que ele utiliza: [20]

- **Métodos estatísticos:** também conhecidos como método de aproximação de entropia, utilizam as probabilidades de ocorrência dos símbolos em um fluxo de dados e alteram sua representação, de modo que eles passem a ser representados por símbolos menores. Exemplos: codificação Huffman e codificação aritmética;
- **Métodos baseados em dicionário:** usam dicionários ou outras estruturas similares de forma a eliminar repetições de símbolos. Exemplos: método LZ77 e método LZ78;
- **Transformações:** são métodos que, por si só, não comprimem os dados, mas são capazes de transformar dados que não seriam comprimidos ou seriam pouco comprimidos pelos métodos normais, em dados que podem ser mais facilmente comprimidos pelos mesmos métodos. Exemplos: transformada discreta de cosseno, utilizado pelos padrões MPEG e JPEG (compressão com perda de dados) e o método de Burrows-Wheeler (compressão sem perda) [6].

Muitos formatos de compressão utilizados atualmente fazem uso de mais de um algoritmo de compressão, sendo a saída de um desses algoritmos usada como

entrada de outro, procedimento repetido sucessivas vezes, até que todos os algoritmos tenham sido utilizados. Este método é chamado de compressão em cascata e permite que altas taxas de compressão sejam alcançadas, em detrimento do tempo de execução.

2.1.1 Algoritmo de Huffman

Em 1952, David A. Huffman propôs uma codificação que faz uso das probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido, para determinar códigos de tamanho variável que representarão cada símbolo, de modo que os símbolos com maior probabilidade de ocorrência sejam codificados com o menor código possível [14]. Os símbolos correspondem a cada sequência de bits de tamanho fixo do arquivo de entrada e possuem uma frequência que representa o número de vezes que este símbolo aparece em um determinado arquivo, sendo esta frequência diretamente proporcional à probabilidade da ocorrência do símbolo. Este método é sem perda, ou seja, todos dados do arquivo de entrada são codificados em dados menores na hora da compressão e, após a descompressão, esses dados são totalmente reconstruídos, sem nenhuma alteração [25]. A importância deste algoritmo de codificação é grande, uma vez que muitos tipos de compressão de dados utilizados em larga escala o utilizam como método único ou como um dos métodos da compressão em cascata. São exemplos o formato de arquivo GZIP (GNU Zip), geralmente utilizado para comprimir arquivos do sistema operacional Linux, o formato PKZIP e o formato JPEG [20] para compressão de imagens.

Seja P(a) uma abreviação para P(X = a), onde X é uma variável aleatória que pode assumir os valores dos símbolos e A = $\{a_1, a_2, \ldots, a_n\}$ o alfabeto da mensagem que será comprimida, onde o alfabeto está ordenado de forma decrescente de acordo com as suas frequências, ou seja:

$$\mathrm{P}(a_1) \geq \mathrm{P}(a_2) \geq ... \geq \mathrm{P}(a_n),$$

deseja-se construir um conjunto de códigos ótimo onde:

$$\mathrm{L}(a_1) \leq \mathrm{L}(a_2) \leq ... \leq \mathrm{L}(a_n),$$

com L(a) representando o comprimento do código associado ao símbolo a. Se esta condição for satisfeita durante a codificação, o símbolo mais frequente no arquivo será substituído pelo código de menor comprimento, garantindo assim uma codificação Huffman ótima.

Após ter a contagem da frequência de cada símbolo do arquivo, é gerada uma árvore binária que indica a codificação de cada um deles. Para isto, são criadas inicialmente n árvores, uma para cada símbolo encontrado no arquivo de entrada, com apenas um nó (raiz), que representa o símbolo e sua probabilidade de ocorrência. Sucessivamente, selecionam-se as duas árvores cujas raízes têm a menor frequência dentre todas e cria-se um novo nó raiz cuja frequência deste é igual à soma das frequências das raízes das árvores selecionadas e estas viram filhos desta nova raiz (com a raiz de maior frequência à esquerda), como demonstrado na figura 2.1.

Ao final obtém-se apenas um nó raiz que representa a árvore de Huffman gerada para a codificação dos símbolos. Para achar o código respectivo de cada símbolo, basta caminhar na árvore da raiz até os nós-folha acrescentando um 0 ao código, caso o próximo nó a ser visitado seja o nó esquerdo, ou um 1, caso seja o direito. Este método garante que os símbolos mais frequentes fiquem com os códigos menores, uma vez que as folhas dos níveis mais altos correspondem aos símbolos que têm maior probabilidade de ocorrência.

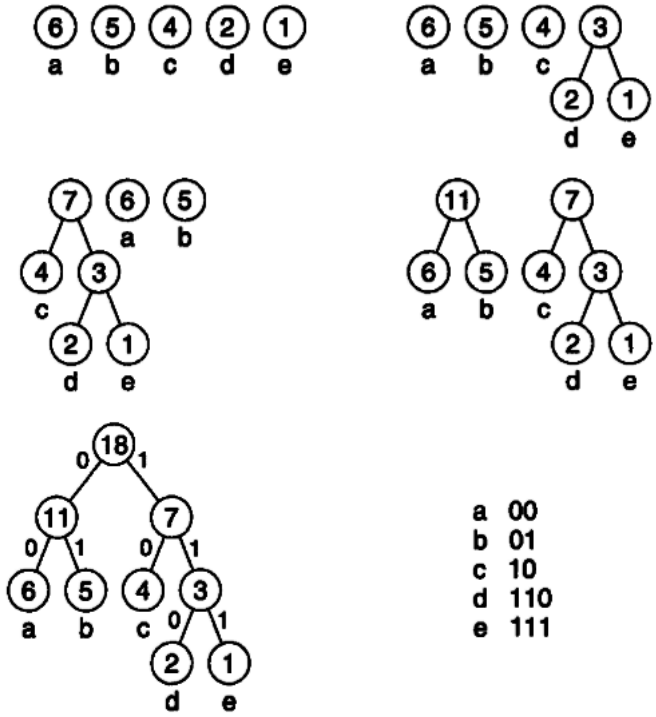

Figura 2.1: Formação da árvore de Huffman e codificação gerada para um determinado conjunto de símbolos

No caso da figura 2.1, se o arquivo original a ser comprimido utilizar a codificação ASCII, serão usados 8 bits para representar cada caracter. Após a compressão, por ser um arquivo com pouca variedade de símbolos (apenas 5 símbolos diferentes), o símbolo mais frequente será representado por apenas 2 bits, enquanto que os menos frequentes serão representados por no máximo 3 bits. Com o crescimento da variedade dos símbolos no arquivo de entrada a ser comprimido, a árvore de Huffman fica cada vez maior, fazendo com que as codificações sejam maiores e algumas podem ter uma codificação maior do que o próprio símbolo original, o que no final é compensado pelos símbolos mais frequentes, que possuem codificações bem pequenas, conseguindo gerar no final um arquivo comprimido menor do que o original.

Para descomprimir um arquivo, é necessário que o *decoder* reconstrua essa árvore a partir das codificações que o compressor calculou (ou uma estrutura equivalente). Depois, o descompressor lê bit a bit o arquivo comprimido, percorrendo a árvore, desde a raiz, visitando o filho à esquerda de um nó, caso tenha lido um bit 0, ou o filho à direita, no caso de um bit 1. Ao chegar em um nó folha, significa que o descompressor encontrou um símbolo, e o mesmo deve ser escrito no arquivo que está sendo reconstruído. Ao encontrar um símbolo, o *decoder* volta a percorrer a árvore, a partir do nó raiz, continuando a leitura dos bits do arquivo comprimido. Isso é feito sucessivas vezes, até que ele tenha consumido todos os bits do arquivo comprimido.

Um aspecto importante deste método de codificação é que um código nunca será prefixo de outro e isto é facilmente demonstrado, pois os símbolos estão sempre nas folhas da árvore, fazendo com que o caminho que leva até um símbolo seja único e não ambíguo. Sendo assim, não é necessário utilizar uma sequência de bits como separador entre os caracteres, pois no momento da descompressão basta ler bit a bit o arquivo comprimido e percorrer a árvore até chegar em uma folha e então refazer o caminho até chegar no final do arquivo, como uma máquina de estados.

Este método gera um pequeno *overhead*, uma vez que é necessário anexar ao arquivo comprimido um cabeçalho contendo a codificação que foi utilizada pelo compressor. Por isto, este método é aconselhado para arquivos grandes, pois com arquivos pequenos este *overhead* faz com que o tamanho final do arquivo compactado fique maior que o do arquivo original. Algumas variações da codificação Huffman original são propostas justamente para diminuir o tamanho deste cabeçalho, fazendo com que ele seja reconstruído a partir de apenas alguns dados.

A codificação de Huffman foi escolhida como algoritmo de compressão para este trabalho por se tratar de um algoritmo sem perda de dados, que pode ser utili-

zado em qualquer tipo de arquivo, e proporciona boas taxas de compressão. Similar a outros métodos estatísticos de compressão de dados, o algoritmo de Huffman requer duas leituras do arquivo de entrada, exigindo maior esforço para lidar com as operações de I/O.

2.1.2 Aplicação sequencial para compressão/descompressão de dados usando o algoritmo de Huffman

Esta seção tem como objetivo detalhar o fluxo de execução de uma aplicação que realiza a compressão e a descompressão de um arquivo, utilizando o algoritmo de Huffman para codificação de bytes, o que significa que cada byte (sequência de 8 bits) do arquivo de entrada terá uma sequência única de bits para representá-lo no arquivo comprimido. Como explicitado anteriormente, os bytes mais frequentes serão representados pelas codificações de tamanhos menores e, assim, espera-se que o arquivo comprimido tenha um tamanho menor que o original.

Neste contexto, podem ser destacados estes dois fluxos principais: um fluxo para a compressão e um para a descompressão. Eles se interligam pois a saída do fluxo de compressão precisa ser a entrada do fluxo de descompressão, porém eles são executados em momentos diferentes, com objetivos diferentes e, às vezes, em computadores diferentes. No exemplo de um usuário que deseja comprimir um arquivo para enviá-lo via rede para outra pessoa, diminuindo-se assim, o tempo necessário para a transferência, primeiro ele deve executar o fluxo da compressão em sua máquina. Ao obter o arquivo comprimido, ele envia via rede para outro usuário, que precisa executar o fluxo de descompressão, obtendo-se assim um arquivo igual ao original.

O fluxo de compressão é iniciado com o usuário escolhendo a sequência de

bytes que deseja comprimir, representada por um arquivo. Primeiramente, a aplicação deverá fazer a contagem de quantas vezes cada byte aparece na sequência de bytes especificada como entrada, que geralmente fica armazenada em um disco rígido, um armazenamento com desempenho de acesso baixo. Após percorrer o arquivo inteiro fazendo a contagem dos bytes, o fluxo de compressão continua com a aplicação direta do algoritmo de Huffman, onde será montada uma árvore binária baseada na frequência de cada um dos bytes presentes no arquivo. Conforme explicado, a partir da árvore montada, a aplicação deve percorrê-la do nó raiz até cada uma de suas folhas para descobrir por qual sequência cada um dos bytes da entrada será substituído no arquivo comprimido. De posse destas informações, a aplicação deve guardar esta codificação utilizada em um arquivo, para que na hora de descomprimir, ela saiba qual byte corresponde à sequência de bits encontrada no arquivo comprimido, e depois, ela precisa percorrer o arquivo de entrada, escrevendo para cada byte lido, a sequência de bits correspondente no arquivo comprimido. O resultado final do fluxo de compressão é, então, um arquivo comprimido e um cabeçalho que especifica qual codificação foi utilizada nesta compressão.

No momento de utilizar o arquivo, é iniciado o fluxo de descompressão, que tem como entrada o arquivo comprimido e a codificação utilizada. Como a codificação já está pronta, este fluxo é um pouco mais simples que o de compressão, uma vez que ele não precisa tirar medidas estatísticas sobre o arquivo. Primeiramente, a aplicação precisa ler os dados da codificação utilizada para a compressão, para depois montar a árvore de Huffman que gerou esta codificação. Uma vez que a aplicação já conhece esta árvore, ela só precisa percorrer o arquivo comprimido, bit a bit, caminhando na árvore até encontrar um nó folha, que corresponde ao byte do arquivo original. Este byte deve então ser escrito no arquivo que está sendo restaurado, e, ao terminar de percorrer o arquivo comprimido repetindo estes passos, será obtido um arquivo exatamente igual ao original.

A partir disto, três passos principais podem ser destacados no fluxo de compressão de um arquivo: (i) uma leitura completa do arquivo de entrada para contabilizar a frequência de ocorrência de cada símbolo encontrado; (ii) a criação da árvore binária de codificação; e (iii) outra leitura completa do arquivo de entrada para gerar o arquivo de saída com os símbolos codificados.

Já o fluxo de descompressão pode ser dividido em apenas duas partes: (i) a leitura do cabeçalho do arquivo comprimido que contém as informações sobre a codificação utilizada; e (ii) a descompressão dos dados, lendo uma sequência de bits do arquivo comprimido e escrevendo, no arquivo restaurado, o byte correspondente.

Um fato importante a ser levado em consideração neste tipo de aplicação é que ela funciona a nível de bits. O arquivo comprimido é composto pela concatenação de sequências de bits com tamanhos variáveis, e não são incluídos marcadores no arquivo de saída indicando onde começa ou termina cada uma destas sequências. Desta forma, torna-se complexo dividir o arquivo comprimido em blocos uma vez que não é possível determinar onde começa e onde termina uma sequência.

Como a compressão é feita a nível de bits e a menor unidade que pode ser armazenada em um arquivo de saída é um byte, pode ocorrer do último byte escrito no disco não ter todos os bits utilizados e, portanto, eles seriam preenchidos com bits aleatórios, o que fará com que o descompressor tente interpretar estes bits aleatórios como símbolos do arquivo original. Possíveis soluções para isto envolvem escrever, junto do arquivo comprimido, a quantidade de bits utilizados no total, ou utilizar um símbolo como marcador de fim de arquivo. Este marcador entraria durante a formação da árvore de Huffman e teria uma codificação. O descompressor, ao ler o total de bits estipulado, ou encontrar a codificação correspondente ao marcador de fim de arquivo, deverá encerrar a execução do fluxo de descompressão.

2.2 Modelo MapReduce e Apache Hadoop

O MapReduce é um modelo de programação desenvolvido para simplificar o processamento paralelo de grandes quantidades de dados. O modelo MapReduce foi inicialmente introduzido pela Google no ano de 2004, inovando a maneira como se armazenavam e tratavam suas grandes quantidades de dados, constituídos, na época, basicamente, pelas páginas indexadas em sua base de pesquisa. Posteriormente, o modelo MapReduce se popularizou e passou a ser utilizado por diversas outras empresas, cada uma com sua própria implementação. O Hadoop é um *framework* para processamento de grandes volumes de dados, inicialmente criado para fornecer aos programadores facilidades para a programação paralela e distribuída baseada no modelo MapReduce.

2.2.1 Modelo MapReduce

Com o crescimento da quantidade de dados de todos os tipos em suas bases, a Google viu a necessidade de adotar modelos de computação paralela e distribuída para que os dados fossem processados em tempo viável, e, na época, o modo de paralelizar, distribuir e lidar com as falhas neste tipo de computação foi o grande desafio encontrado por pesquisadores da empresa. Para tentar vencer este desafio, estes pesquisadores propuseram um modelo de programação paralelo e distribuído chamado de MapReduce, juntamente com o *framework* de mesmo nome, onde os usuários deste *framework* precisavam apenas se preocupar com a implementação de uma função de *map* (para mapeamento dos dados), e uma função de *reduce* (para redução destes dados mapeados), enquanto o particionamento do arquivo de entrada e a tolerância a falhas ficavam por conta da implementação do próprio *framework*. Todas estas facilidades foram pontos que contribuíram para o sucesso deste modelo [8].

A abstração MapReduce foi inspirada em operações presentes em algumas linguagens de programação, como Lisp. A maioria das computações que os inventores deste modelo precisavam realizar aplicava uma operação de mapeamento para que fossem encontrados pares intermediários de chave/valor, e depois uma operação de redução dos valores que tivessem a mesma chave, combinando assim todos os valores intermediários, encontrando um valor final.

2.2.2 Apache Hadoop

Baseado no modelo MapReduce do Google, surgiu o Hadoop, marca registrada da Apache, que representa a implementação de um projeto inicialmente desenvolvido pela Yahoo!. Trata-se de um *framework* de código fonte aberto, desenvolvido em Java, para permitir a execução de aplicações paralelas que trabalham sobre dados massivos baseadas no modelo MapReduce, usando um conjunto de computadores conectados por uma rede. A arquitetura do Hadoop é composta por *daemons* principais e secundários, onde os principais são responsáveis pela gerência do cluster, a interface com o cliente e escalonamento das tarefas, e os secundários são utilizados para armazenar e processar os dados. Um *daemon* é um aplicativo que roda em segundo plano em um sistema operacional, neste caso executando operações de comunicação ou monitoração. Esta abstração de uso de *daemons* permite que, em um cluster, um mesmo nó possa ser responsável pelas tarefas de gerência do cluster e de processamento e armazenamento de dados.

O framework Hadoop possui um conjunto de componentes para armazenamento de dados distribuídos entre os nós (chamado *Hadoop Distributed File System*), e outro conjunto de elementos para o processamento dos dados, os quais são totalmente independentes entre si. Ele foi projetado para que fosse escalável desde clusters de apenas um nó, a clusters de milhares de máquinas, onde cada uma oferece

processamento e armazenamento de dados. Este tipo de armazenamento distribuído é eficiente para arquivos muito grandes [15].

Em suas versões iniciais, a parte responsável pelo processamento dos dados do Hadoop podia ser dividida em três camadas:

- API MapReduce: interface usada pelos programadores para escreverem suas aplicações baseadas no modelo MapReduce;
- Componentes de execução: responsáveis pela execução das tarefas de mapeamento, ordenação e redução dos *jobs* MapReduce;
- Componentes de infraestrutura: responsáveis pela infraestrutura necessária para a execução das aplicações, gerência de recursos, escalonamento de tarefas, dentre outros.

Esta segregação facilitou a vida dos usuários do *framework*, uma vez que eles precisavam apenas se preocupar em utilizar a API de forma correta, deixando a responsabilidade da execução, gerência e escalonamento de tarefas a cargo do *framework*.

Em suas versões iniciais, que permitiam apenas a execução de *jobs* MapReduce, a infraestrutura do *framework* Hadoop possuía dois componentes principais. Um deles era o *JobTracker*, que era executado apenas em um nó e era responsável por atender às requisições dos usuários para a execução de aplicações MapReduce, escaloná-las entre os nós de processamento e gerenciar o estado da execução destas aplicações. Ele se comunicava com os nós de processamento através do outro componente, o *TaskTracker*, que possuía uma instância em execução em cada nó de processamento. Ele era responsável pela execução das operações de *map* (mapeamento), *shuffle* (ordenação) e *reduce* (redução) dos *jobs* MapReduce que o *JobTracker* escalonava. Cada *TaskTracker* possuía uma configuração que indicava quantas

destas operações ele poderia executar de forma simultânea, dependendo dos recursos de processamento e memória RAM que o nó oferecia. Ao terminar a execução de uma dessas operações, o *TaskTracker* avisava ao *JobTracker*, que tomava a decisão sobre como prosseguir com a execução da aplicação.

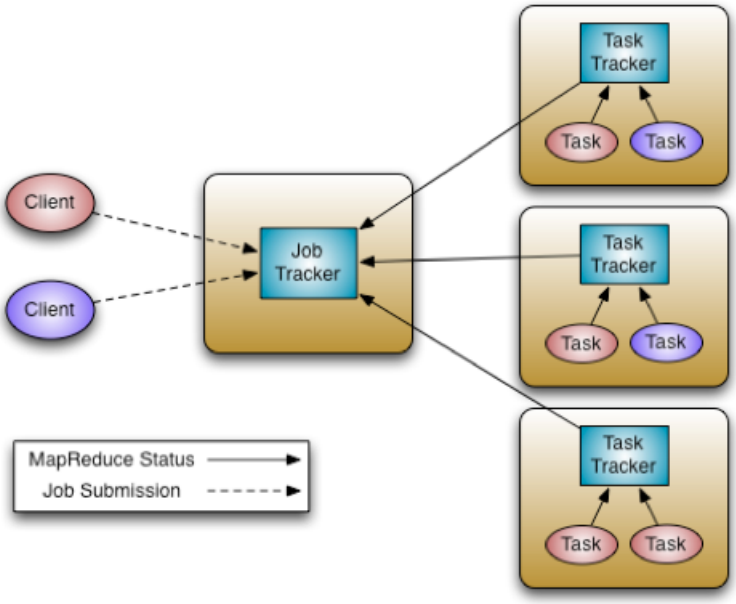

Figura 2.2: Exemplo de funcionamento da parte responsável pelo processamento de tarefas MapReduce nas versões iniciais do Hadoop[2]

A figura 2.2 demonstra a execução dos *daemons* responsáveis pelo processamento de diferentes *jobs* MapReduce, divididos em tarefas (*tasks*), e o processamento destas tarefas nos diferentes nós de processamento.

O sistema de arquivos distribuído HDFS, que possui a mesma arquitetura nas versões antigas e novas do Hadoop, funciona de forma análoga aos nós de processamento. No nó de gerência, o *NameNode* é o responsável por manter toda a estrutura de diretórios e os metadados de cada arquivo armazenado, que pode ter seus dados divididos entre os nós de armazenamento. O nó de gerência do sistema

[2]Fonte da imagem: http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

de arquivos se comunica com os nós de armazenamento através do *DataNode*, que possui uma instância em execução em cada um dos nós de armazenamento. Um *DataNode* utiliza o disco da máquina onde é executado para que sejam armazenadas as partes de um arquivo no HDFS. Quando um cliente HDFS requisita ao sistema de arquivos do Hadoop o armazenamento de um arquivo, ele se comunica com o *NameNode* que, por sua vez, se comunica com os *DataNodes* para selecionar em quais nós o arquivo será armazenado. O arquivo é dividido em partes, de forma que todas elas tenham um tamanho limite. Cada uma destas partes é armazenada em um determinado número de nós, para que haja redundância de dados e um arquivo possa ser acessado mesmo que alguns nós de armazenamento apresentem falhas. O tamanho limite de cada parte e o número de nós onde uma parte será replicada são exemplos de configurações do sistema de arquivos especificadas pelo administrador do ambiente. Ao tentar acessar um arquivo, um cliente também se comunica diretamente com o *NameNode*, recebendo uma lista com os nós que armazenam cada parte do arquivo.

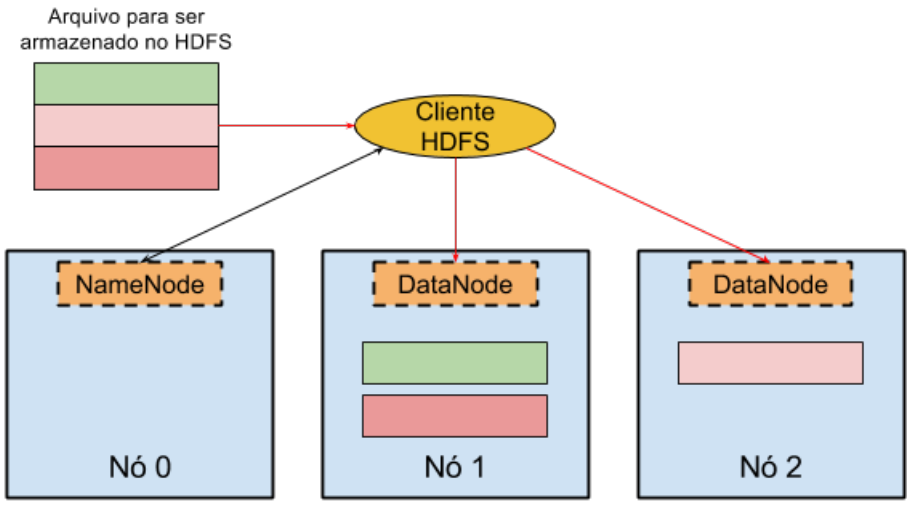

Figura 2.3: Exemplo de funcionamento do HDFS

A figura 2.3 mostra o processo de armazenamento de um arquivo no sistema de arquivos distribuído do Hadoop. Inicialmente, o cliente HDFS se comunica com o

NameNode requisitando o armazenamento de um arquivo. O *NameNode* armazenará os metadados deste arquivo e do particionamento que deverá ser feito, e enviará uma resposta ao cliente com os dados dos nós que armazenarão as partes deste arquivo. O cliente, então, se comunica diretamente com os *DataNodes* destes nós, enviando os pedaços do arquivo correspondentes, conforme explicitado na resposta do *NameNode*.

Inicialmente projetado para dar suporte apenas a aplicações do modelo MapReduce, este *framework* apresentou diferenciais que permitiram seu sucesso e impulsionaram sua utilização em larga escala: as automatizações da comunicação entre os nós, da divisão do arquivo de entrada, do escalonamento das tarefas e também da gerência e da garantia de tolerância a falhas. O suporte a falhas é um diferencial também por tolerar erros físicos nos nós de um cluster, fazendo com que uma aplicação possa continuar executando normalmente. Isto permite que sejam usados nós com hardware de baixo custo e, consequentemente, baixa qualidade para a montagem de um cluster para executar aplicações paralelas e distribuídas que rodem através deste *framework*.

Nas versões iniciais do Hadoop, que só dão suporte ao modelo MapReduce, para que um *job* MapReduce seja executado, primeiramente o arquivo de entrada precisa ser armazenado no sistema de arquivos distribuído do Hadoop, que o divide automaticamente entre os nós de armazenamento. Quando o *JobTracker* recebe do cliente a requisição de execução de um *job* MapReduce, ele faz uma consulta ao *NameNode* para saber em quais nós estão armazenadas as partes deste arquivo. De posse desta informação, o *JobTracker* dispara um container de *map* para cada parte do arquivo, dando preferência para alocá-lo no nó onde está armazenada tal parte, mas ele pode ser alocado em outro nó, caso este nó já tenha muitas tarefas alocadas, ou pode enfileirá-lo para ser executado posteriormente, caso não haja nenhum nó disponível. O usuário deve especificar como será feito o particionamento de um

arquivo de entrada nos pares de chave/valor, escolhendo uma classe que fará esta tarefa, chamada de particionador.

Um container é um objeto executável Java disparado em cada um dos nós escolhidos para realizar o processamento. A API do Hadoop possui duas classes abstratas — *Mapper* e *Reducer* — que o usuário do *framework* deve estender para implementar suas próprias classes de *map* e de *reduce*, as quais servirão como base para os containers de mapeamento e de redução, respectivamente. Cada uma destas classes possui três métodos principais que podem ser sobrescritos pelo usuário: *setup*, *map* ou *reduce* (dependendo da classe que foi estendida), e *cleanup*. Ao iniciar um container, o *framework* executa o método *setup*, que inicializa dados do container. Em seguida, os métodos *map* ou *reduce* são chamados para processar cada par <chave, valor> sucessivas vezes, até que não existam mais pares para serem processados. Quando esgotados estes pares, o método *cleanup* é executado, encerrando a execução do container. Com isto, cabe ao usuário implementar os métodos que desejar, da maneira que achar mais conveniente para a sua aplicação.

Um container de *map*, ao ser executado, receberá como entrada uma sequência de pares de chave/valor, de acordo com o formato especificado pelo particionador escolhido. Durante sua execução, o *framework* requisita pares de chave/valor ao particionador e, para cada par obtido, ele executa o método *map*, utilizando tal par como entrada. Este processo é repetido até que seja totalmente consumido o bloco do arquivo de entrada pelo qual este container é responsável. Durante o processamento destes pares da entrada, o container *map* pode emitir outros pares de chave/valor, em formato especificado pelo usuário. Quando o container encerrar sua execução, o *framework* ordenará e agrupará estes pares de chave/valor pela chave, gerando um par com esta chave e uma coleção de valores para ela. Em seguida, serão disparadas os containers de *reduce*, cuja quantidade pode ser especificada pelo usuário. Cada um deles também executa os métodos *setup* ao ser iniciado, *reduce* para cada par

de chave/valor atribuído a ele, e *cleanup* ao final de sua execução. Durante este processo, o container pode emitir novos pares de chave/valor, que representarão o resultado final do *job*. O *framework*, ao receber estes pares emitidos pelos containers de redução, os escreverá no sistema de arquivos distribuído, de acordo com o formato de saída escolhido.

Após a ordenação dos pares de saída do mapeamento, a distribuição padrão destes pares entre os containers de redução acontece de forma intercalada. Por exemplo, caso existam dois containers *reduce* e cinco chaves diferentes, o primeiro ficará com as chaves nas posições 1, 3 e 5 após a ordenação, enquanto que o segundo ficará com as chaves nas posições 2 e 4. Um usuário pode optar por implementar um particionador intermediário que decide qual container de redução deve processar aquela chave.

Os objetos responsáveis por realizar as tarefas de particionamento do arquivo de entrada em pares de chave/valor, o mapeamento destas chaves, a ordenação da saída da função de *map* e a redução devem ser explicitados pelo usuário durante o desenvolvimento da aplicação. O Hadoop já possui alguns tipos pré-definidos para isto, mas o usuário pode implementar e optar pela utilização de seus próprios tipos, de acordo com sua necessidade.

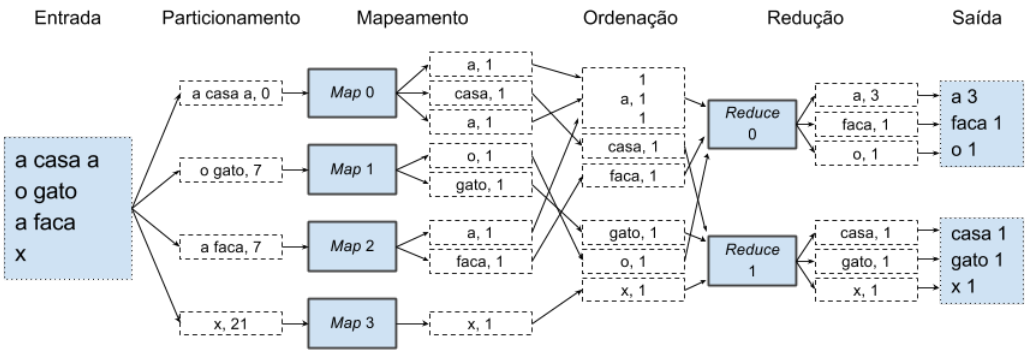

Figura 2.4: Exemplo de aplicação baseada no modelo MapReduce

A figura 2.4 mostra uma aplicação para a contagem de palavras em um determinado arquivo de entrada como exemplo do modelo MapReduce. Nela, pode-se perceber com clareza o funcionamento das funções intermediárias, executadas automaticamente pelo *framework* Hadoop e das funções de mapeamento e de redução implementadas pelo usuário, onde todas utilizam tipos nativos do Hadoop. Inicialmente, um arquivo de texto é armazenado (distribuído entre os nós no HDFS) e quando o *JobTracker* recebe do cliente a requisição de execução de um *job* MapReduce para a contagem de palavras deste arquivo, ele faz uma consulta ao *NameNode* para saber em quais nós estão armazenadas as partes deste arquivo. Para este exemplo, foi utilizado como particionador o tipo nativo do Hadoop *TexitInputFormat* que, juntamente com a sua implementação de um *RecordReader*, é responsável por dividir um arquivo de texto em pares de chave/valor, onde o valor é do tipo *TextWritable* e corresponde a cada linha do arquivo, e a chave é do tipo *LongWritable* e representa o *offset*, em bytes, desta linha dentro do arquivo de entrada. O particionador fornece todos estes pares para a tarefa de *map* que, ao ser executada, recebe como entrada esta sequência de chave/valor, processa-a e emite um par de chaves do tipo *TextWritable* e valores do tipo *IntWritable*, onde cada chave é uma palavra encontrada e o valor é sempre 1, indicando uma ocorrência daquela palavra. Estes pares emitidos serão ordenados pela função de ordenação do próprio Hadoop, de acordo com as suas chaves, e distribuídos entre os dois containers de redução pelos particionadores intermediários, que irão somar todos os valores para uma mesma chave, emitindo como saída cada uma das chaves de entrada e seu somatório final. Depois, todos estes resultados são escritos no HDFS para que o usuário possa acessar esta contagem final.

```
public static class WordCountMap extends Mapper<LongWritable,
   Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context
       context) {
```

```
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

Listagem 2.1: Código fonte em Java da classe *map* do exemplo da contagem de palavras. Fonte: http://wiki.apache.org/hadoop/WordCount

```
public static class WordCountReduce extends Reducer<Text,
   IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context) {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }

        context.write(key, new IntWritable(sum));
    }
}
```

Listagem 2.2: Código fonte em Java da classe de *reduce* do exemplo da contagem de palavras. Fonte: http://wiki.apache.org/hadoop/WordCount

Os códigos 2.1 e 2.2 representam uma implementação possível para as classes de mapeamento e redução do exemplo da contagem de palavras citado acima. Nestes exemplos, ficam evidentes as tarefas do usuário de especificar os tipos e implementar as principais classes necessárias para a execução de um *job* MapReduce. O código 2.3 demonstra como uma aplicação Java deve especificar as configurações e requisitar a execução deste *job* MapReduce para a contagem de palavras do arquivo. Para uma melhor visualização destes códigos, foram omitidas as declarações das exceções que cada método pode lançar.

```
public static void main(String[] args) {
    Configuration conf = new Configuration();
```

```
    Job job = new Job(conf, "wordcount");

    job.setInputFormatClass(TextInputFormat.class);

    job.setMapperClass(WordCountMap.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setReducerClass(WordCountReduce.class);

    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path("/Files/
       wordcount_example"));
    FileOutputFormat.setOutputPath(job, new Path("/Files/
       wordcount_output"));

    job.waitForCompletion(true);
}
```

Listagem 2.3: Código fonte em Java que requisita a execução de um *job* MapReduce para a contagem de palavras. Fonte: http://wiki.apache.org/hadoop/WordCount

No código 2.3, o usuário especifica, nesta ordem: o tipo do arquivo de entrada do *job* (particionador), a classe que representa o container de mapeamento, os tipos dos pares de chave/valor intermediários (saída dos containers *map* e entrada dos containers *reduce*), a classe que representa o container de redução, o formato em que o *framework* deve escrever os pares emitidos pelos containers de redução nos arquivos de saída e o caminho dos arquivos de entrada e de saída. Além disto, também é possível especificar o número de containers de *reduce* que o usuário deseja que sejam executados.

A divisão de um arquivo ao ser armazenado no HDFS é feita a nível de bytes, de forma que cada parte do arquivo tenha um tamanho máximo, que pode ser definido nas configurações do *framework*. Com isso, ao armazenar um arquivo

de texto, uma palavra (e consequentemente uma linha) pode ser dividida de forma que cada parte dela fique em um pedaço diferente do arquivo, gerando duas palavras novas. Para contornar este problema, o particionador de texto do próprio Hadoop (*TextInputFormat*), juntamente com o seu *RecordReader*, identificam casos deste tipo e os tratam, acessando o resto da linha na outra parte do arquivo, mesmo que ela não esteja armazenada fisicamente naquele nó. Isto ocorre de forma totalmente transparente para as funções de mapeamento, que receberão como valor as linhas completas, simplificando a implementação dos containers *map* e *reduce*. A figura 2.5 demonstra como este processo é feito e como os pares são entregues aos containers de mapeamento.

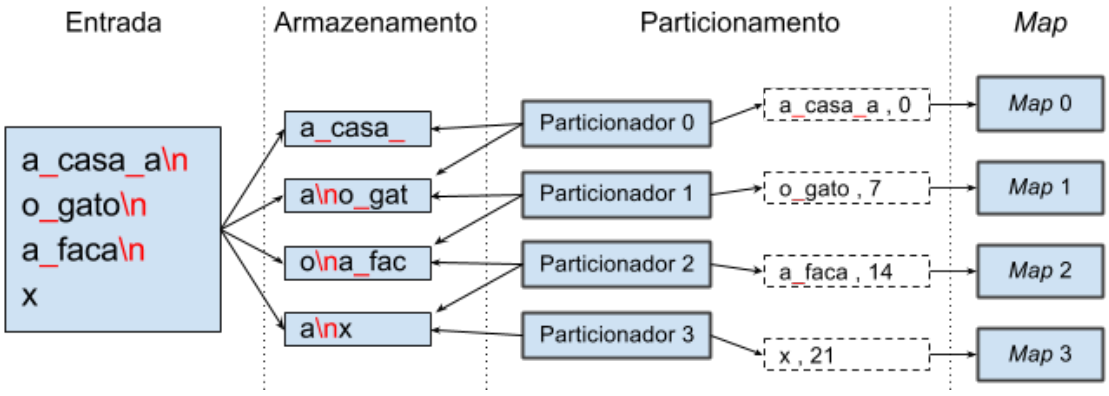

Figura 2.5: Exemplo de funcionamento do particionador de texto do próprio Hadoop

Neste exemplo, o sistema de arquivos está configurado para dividir o arquivo em blocos de no máximo 7 bytes. Estes blocos são armazenados em nós escolhidos pelo sistema de arquivos. Cada particionador, ao detectar que há uma parte da linha em outro nó, realiza uma requisição para obter o restante da linha, fornecendo assim, linhas inteiras ao container de *map*.

Assim, uma das grandes vantagens do modelo MapReduce oferecido pelo Hadoop é a praticidade oferecida ao programador, que não precisa se preocupar com tarefas complicadas, como particionar o arquivo de entrada entre os nós que

irão rodar a aplicação, escalonar as tarefas, gerenciar a comunicação entre os nós, nem implementar técnicas de tolerância a falhas. Apesar de todas estas facilidades, vale ressaltar que o programador precisa escrever sua aplicação seguindo sempre esse modelo, i.e., usando funções *map* e *reduce*, o que nem sempre é possível ou trivial.

2.2.3 Hadoop YARN

Todas as facilidades especificadas anteriormente contribuíram para o grande sucesso do Hadoop. Porém, nem todo o tipo de tarefa pode ser facilmente expressa pelo modelo MapReduce, como por exemplo o processamento de grafos e modelos iterativos, onde as unidades de processamento precisam se comunicar entre si.

Com isto, os desenvolvedores do *framework* reescreveram seus principais componentes de forma a possibilitar a execução de aplicações paralelas que não seguem o modelo MapReduce. Com isto, os usuários podem fazer uso dos recursos do *framework* para gerência de recursos, escalonamento de tarefas e administração do cluster, e desenvolver seu próprio modelo de programação. No modelo MapReduce, a orquestração das tarefas é feita pelo componente Hadoop MapReduce, que se comunica com o componente do *framework* responsável pela gerência. Já em aplicações que não seguem esse modelo, cabe ao usuário do *framework* desenvolver o componente que fará a orquestração da sua própria aplicação. Isto elimina algumas das facilidades às quais o *framework* se comprometia a oferecer, porém permite que um leque maior de aplicações o utilize.

Nas primeiras versões lançadas, o Hadoop possuía dois componentes principais: o sistema de arquivos distribuído HDFS, responsável pelo armazenamento dos dados, e o *framework* responsável pelo processamento, que realizava a gerência de recursos, o escalonamento de tarefas, e fornecia uma biblioteca para a implementa-

ção e execução de aplicações que seguem o modelo MapReduce. Nas suas versões mais recentes, mais especificamente nas versões acima da 2.0, o Hadoop alterou um pouco a sua arquitetura, de forma que o sistema de arquivos distribuído permaneceu igual, porém houve uma segregação na parte do *framework* responsável pelo processamento: foi criada uma arquitetura nova, chamada de YARN (*Yet Another Resource Negotiator*), separando a parte responsável pela gerência e escalonamento da parte responsável pela execução das tarefas MapReduce, como demonstrado pela figura 2.6.

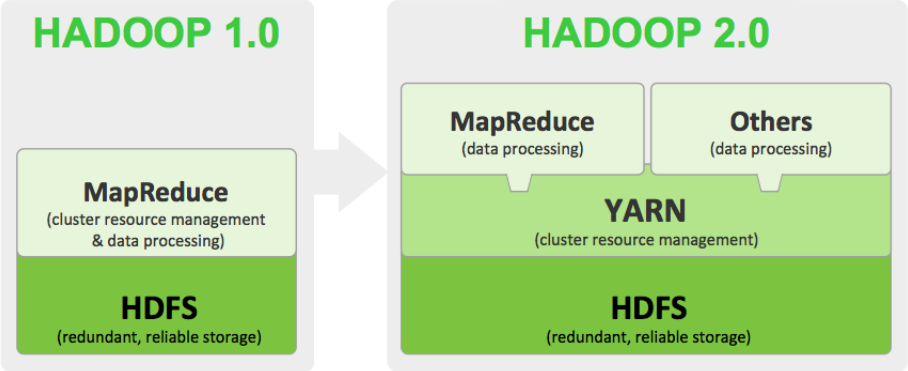

Figura 2.6: Comparação entre a arquitetura do Hadoop 1.x e Hadoop 2.x[3]

Com isto, os componentes da arquitetura YARN passaram a ser os responsáveis pela gerência de recursos do cluster e o Hadoop MapReduce passou a ser um *framework* de programação separado, que utiliza os componentes do Hadoop YARN, fazendo a interface entre esses componentes e as classes da aplicação do usuário.

Esta separação é importante, pois permite que um usuário crie sua aplicação, que não precisa seguir o modelo MapReduce, de forma que ela comunique-se diretamente com os componentes do Hadoop YARN que vão gerenciar os recursos do cluster e monitorar o estado da execução da aplicação nos nós. Isto permite, então, que outros *frameworks* como o Hadoop MapReduce sejam desenvolvidos, facilitando ou otimizando a criação de outros tipos de aplicações distribuídas, de forma que elas utilizem da melhor forma os recursos disponíveis em um cluster YARN. Além disto,

tais *frameworks* podem fornecer suporte à execução de programas escritos em linguagens não suportadas nativamente pelo Hadoop.

Os principais exemplos de *frameworks* que executam sobre a arquitetura YARN, além do Hadoop MapReduce, são:

- **Apache Giraph:** implementação de código fonte aberto que permite o processamento distribuído de grafos com alta escalabilidade. É utilizado atualmente pelo Facebook, para a análise de grafos de redes sociais [3];
- **Apache Storm:** desenvolvido para aplicações que necessitam processar *streams* de dados em tempo real, e
- **Hoya:** do acrônimo *HBase on YARN*, este projeto permite a criação de um cluster para persistência de dados não relacionais, utilizando o HBase.

Um dos principais componentes da arquitetura do Hadoop nas primeiras versões, o *JobTracker*, foi separado em dois *daemons* nas versões mais novas: o *ResourceManager* e o *ApplicationMaster*. O primeiro é único em um cluster, fica em execução no nó de gerência, e é o responsável por gerenciar os recursos do cluster YARN. Já o segundo, é um componente que cada aplicação deverá possuir uma instância, implementada a sua maneira, e será o responsável por requisitar recursos do cluster e escalonar suas tarefas nestes recursos. Um *ApplicationMaster* será executado em um container alocado em algum dos nós de processamento do cluster. Esta separação permitiu que o Hadoop alcançasse alto grau de escalabilidade, uma vez que a tarefa de gerência de recursos passou a ser executada em um outro nó, diferente do nó que executa as tarefas de tolerância a falhas e escalonamento de tarefas [28]; e permitiu, também, que outros *frameworks* pudessem implementar

[3]Fonte da imagem: http://www.tomsitpro.com/articles/hadoop-2-vs-1,2-718.html

seus próprios *ApplicationMasters* para que suas aplicações fossem executadas de forma distribuída sobre o Hadoop.

O *ResourceManager* é um *daemon* que possui interfaces para comunicação com os clientes YARN, os administradores do cluster e os demais nós. Ele troca, constantemente, informações com cada um dos nós de processamento do cluster para saber o estado de cada um deles (quantidade de memória RAM disponível, carga de CPU e etc). Ao receber uma requisição para alocação de algum container, ele decide qual nó será escolhido para atendê-la. Ele também possui uma interface web para que o administrador do cluster possa visualizar informações e gerenciar a execução de aplicações.

Sempre que é feita uma requisição de alocação de recursos ao *ResourceManager*, ele tem o trabalho de escolher algum dos nós para alocar tais recursos e responder um *ContainerLaunchContext* a quem fez a requisição. Um *ContainerLaunchContext* é um objeto que contém todas as informações sobre o container que o *ResourceManager* conseguiu alocar, como espaço de memória, número de núcleos virtuais de CPU, nó onde ele será executado e etc. Quem recebe esta resposta pode especificar neste objeto *ContainerLaunchContext* quais recursos o container deverá acessar localmente no nó onde será executado e qual deverá ser o comando disparado para sua execução. Um container YARN basicamente representa uma coleção de recursos físicos, como memória RAM e núcleos de CPU. O Hadoop YARN aloca as tarefas em containers, onde cada nó pode ter um ou mais containers sendo executados paralelamente, dependendo de seus recursos físicos.

Cada um dos nós de processamento de um cluster YARN possui um *daemon* em execução chamado de *NodeManager*. Ele é responsável por monitorar os recursos do nó, supervisionar a execução dos containers das aplicações e enviar essas informações para o *ResourceManager*. Caso o *ResourceManager* decida por encer-

rar a execução de um container, ele envia esta informação para o *NodeManager*, que realizará este serviço. A figura 2.7 mostra os principais *daemons* e como é feita a comunicação entre os componentes de uma aplicação YARN.

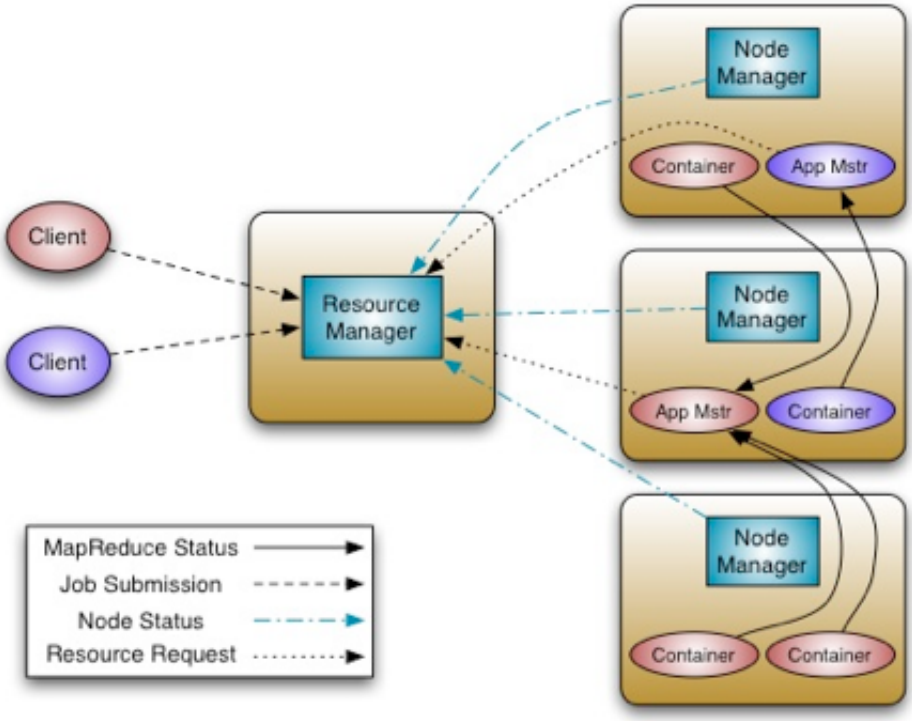

Figura 2.7: Exemplo de funcionamento da parte responsável pelo escalonamento e gerência de recursos no Hadoop YARN[5]

Para que uma aplicação seja executada de forma distribuída em um cluster utilizando a arquitetura YARN, ela necessita de um componente chamado de *YARN Client*. Este componente nada mais é que um objeto Java normal, que não necessita estender nenhuma classe nem implementar nenhuma interface. O cliente YARN é executado na própria máquina onde é chamada a sua execução, que pode estar localizada em qualquer ambiente (interno ou externo ao cluster) e necessita apenas de acesso ao ambiente do cluster, onde está o *ResourceManager*, componente com o qual ele deve se comunicar para poder executar suas tarefas. O cliente, ao ser executado, instanciará objetos com informações sobre o cluster YARN e registrará a aplicação

no cluster, recebendo um identificador para a mesma. Após isso, ele solicitará a alocação de um container ao *ResourceManager* para executar o seu *ApplicationMaster* e, ao receber um *ContainerLaunchContext* como resposta, ele disponibiliza no cluster todos os recursos que a aplicação YARN irá necessitar, como variáveis de ambiente ou arquivos e configura o comando que deverá ser disparado no container para que o *ApplicationMaster* seja executado. Ele pode, também, monitorar o estado da aplicação durante sua execução e o motivo de seu encerramento, pelo identificador que ele recebeu.

O exemplo da aplicação MapReduce para a contagem de palavras de um arquivo foi explicado do ponto de vista do *framework* Hadoop MapReduce que, ao ser executado em um cluster YARN, faz toda esta comunicação com o Hadoop YARN para que seu *job* seja executado, de forma totalmente transparente ao usuário. O usuário, ao desenvolver a aplicação para o *framework* Hadoop MapReduce, gera um arquivo *jar* que é executado em alguma máquina com acesso ao cluster. Este arquivo contém as classes de *map* e de *reduce* implementadas pelo usuário e o código sequencial não distribuído onde estão as configurações do *job* MapReduce e seu comando de inicialização. O *framework* Hadoop MapReduce disponibiliza em tempo de execução seu *ApplicationMaster*, que se comunicará com o *ResourceManager* e com o *NameNode* para escolher os nós onde serão disparados os containers para aquele *job*, monitorará a execução dos containers e retornará uma resposta ao usuário. Sendo assim, o usuário não precisa se preocupar com a parte da implementação da comunicação da sua aplicação com a camada do Hadoop YARN, pois ela é feita pelo próprio Hadoop MapReduce, de forma transparente para o usuário.

[5]Fonte da imagem: http://br.hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

2.3 Clusters Computacionais

Existem diversas formas de se definir o termo cluster, no entanto, a definição mais simples e usual é: um conjunto de computadores interligados por uma rede por onde se comunicam, capazes de funcionar de forma cooperativa para prover uma determinada funcionalidade ou solucionar um determinado problema como se fossem um único e integrado recurso computacional. Esta rede é chamada de rede de interconexão e é utilizada para que os nós troquem mensagens entre si ou acessem as áreas de memória dos outros, dependendo do paradigma utilizado (troca de mensagens ou memória compartilhada). Os clusters são muito utilizados para computação paralela e distribuída.

A configuração de um cluster inclui opções para cada máquina individualmente, valendo citar: a quantidade de memória, o tipo e número de processadores, o tipo de rede que as interligam, o sistema de armazenamento de arquivos empregado e o sistema operacional utilizado.

2.3.1 Classificação

Clusters computacionais podem ser classificados em três grupos básicos, de acordo com seu modo de operação:

- **Balanceamento de carga:** nesse grupo, as máquinas que compõem o cluster processam tarefas individuais simultaneamente, sendo elas escalonadas de forma a manter a carga igual sobre as máquinas. Na maior parte das vezes não existe interação direta entre as diversas máquinas do cluster. Assim, tomando como exemplo um cluster que provê a hospedagem de uma página Web, tem-se

que, quando a página tem um grande número de acessos simultâneos, o responsável pelo balanceamento de carga determina qual das máquinas do cluster deverá responder a uma requisição, evitando assim a sobrecarga em alguma das máquinas.

- **Alta disponibilidade:** uma máquina, ou um subgrupo de máquinas, processa tarefas individuais e reportam o seu estado atual de disponibilidade às máquinas restantes através de um mecanismo conhecido como *keep alive* ou *heart beat*. Caso ocorra uma falha na máquina principal, uma ou mais máquinas restantes assumem as funções desta que falhou, provendo redundância ao sistema de forma transparente ao usuário final. Esse grupo de clusters é útil para aplicações críticas, como servidores de e-mails e de arquivos. Um cluster de balanceamento de carga muitas vezes também é considerado como um cluster de alta disponibilidade;

- **Alto desempenho:** uma tarefa computacionalmente complexa é dividida em pequenas sub-tarefas e escalonada através das máquinas que compõem o cluster, permitindo que elas sejam executadas de forma paralela. Essas, por sua vez, recebem e reportam os respectivos resultados através de mecanismos de troca de mensagem ou de memória compartilhada. Ao final, a união dos resultados parciais fornece o resultado final, com um tempo de execução menor devido à divisão das tarefas. Esse grupo de clusters é muito usado para execução de programas que necessitam de alto poder de processamento, como computação científica e análises financeiras.

Neste trabalho, o enfoque será em Clusters de Alto Desempenho, uma vez que o objetivo é reduzir o tempo de processamento de aplicações para compressão de grandes volumes de dados.

Em um Cluster de Alto Desempenho, cada computador ou máquina que o compõe é chamada de nó. Cada um dos nós de um Cluster HPC tem uma classificação, de acordo com a função que ele exerce dentro do cluster. As três classificações que um nó pode ter dentro de um Cluster HPC são:

- **Nós de acesso:** são os nós responsáveis pela comunicação do mundo externo, onde estão os usuários que desejam executar suas aplicações distribuídas, com o ambiente interno do cluster.
- **Nós de gerência ou controle:** nó, ou conjunto de nós, onde todas as aplicações que compõem a parte operacional do cluster residem. Tais nós são responsáveis por distribuir as tarefas dos usuários no cluster, gerenciar e escalonar uma fila de execução para as aplicações e gerenciar os recursos do cluster. O acesso a estes nós, geralmente, é restrito aos administradores do cluster.
- **Nós de processamento ou computacionais:** nós que processam as tarefas disparadas pelos nós de gerência de forma paralela. Esses nós usualmente não são acessados diretamente pelo usuário, mas apenas através do software de distribuição de tarefas do cluster.

Outro componente importante presente em alguns clusters é chamado de *storage*, que é responsável por prover o cluster com espaço de armazenamento para os arquivos que serão usados como entrada ou saída das aplicações distribuídas executadas no mesmo. Os *storages* utilizados em clusters geralmente fornecem acesso paralelo ao seu espaço de armazenamento através de um sistema de arquivos distribuído, aumentando assim a taxa de leitura e escrita dos arquivos, uma vez que vários nós podem ler ou escrever ao mesmo tempo, sem ter que formar uma fila, como ocorre com vários dispositivos tentando ler um mesmo arquivo em um sistema de arquivos sequencial. Alguns clusters utilizam soluções de *storage* vendidas por

empresas, como é o caso da Panasas, que fornece um conjunto de hardware e um sistema de arquivos próprio. Outros clusters configuram nós separadamente dos nós de computação apenas para fazer o armazenamento, ou então utilizar os próprios nós computacionais para este fim.

Em um cenário de um cluster HPC que tenha todos os componentes citados acima, uma aplicação escrita de forma paralela é executada da seguinte forma: o usuário interage com um nó de acesso do cluster e, caso necessário, armazena o arquivo de entrada para sua aplicação no armazenamento do cluster e, a partir daí, solicita a execução de sua aplicação. Então, o nó de acesso encaminha esta requisição para o nó de controle que escalonará a tarefa deste usuário para ser executada nos nós selecionados, assim que eles estiverem disponíveis. Quando chegar o momento, o nó de controle irá escalonar e executar esta tarefa nos nós computacionais disponíveis, que irão, caso necessário, acessar paralelamente o arquivo de entrada, processar e gerar uma saída para o usuário.

3 TRABALHOS RELACIONADOS

Neste capítulo, destacamos trabalhos da literatura que serviram de referência para este trabalho ou que propõem outras alternativas para melhorar o desempenho de aplicações para compressão de dados. Organizamos esses trabalhos em três grupos: trabalhos que visam a otimização do algoritmo de Huffman especificamente; trabalhos que propõem a paralelização de algoritmos de compressão e trabalhos que usam o framework Haddop para implementar algoritmos de compressão de dados.

3.1 Implementações Otimizadas do Algoritmo de Huffman

Vários autores propõem otimizações para o algoritmo sequencial de Huffman, entre eles Schwartz *et al.* [22] propõem um método de codificação canônico baseado na codificação de Huffman. Gallager [10] e Vitter [29] propõem uma codificação adaptativa de Huffman, a qual pode ser utilizada para a codificação de fluxos em tempo real. A otimização destas versões se deve ao fato de que não há necessidade de percorrer um mesmo fluxo duas vezes, pois se baseiam na criação de uma árvore dinâmica que é alterada conforme as estatísticas do arquivo vão mudando e a codificação é feita já nesta etapa, de acordo com a árvore a cada instante. Hashemian [13] apresenta uma otimização para a busca de um dado em uma árvore de Huffman, otimizando o processo de codificação. O foco deste trabalho não está diretamente relacionado com a otimização do algoritmo de Huffman, mas sim com a contribuição (ganho de desempenho) que a execução desse algoritmo em um ambiente distribuído pode ter sobre um ambiente não-distribuído. Para o estudo realizado neste trabalho, foi escolhido o algoritmo simples de Huffman.

Com relação à paralelização do algoritmo de Huffman, encontramos alguns trabalhos propondo técnicas para paralelizar a montagem da árvore de codificação de Huffman, ou de uma árvore equivalente [26, 17, 2]. Na implementação proposta neste trabalho, focamos na paralelização e distribuição das tarefas diretamente ligadas às operações de entrada e saída dos dados. A montagem da árvore de Huffman permaneceu sequencial. As diferentes propostas de paralelização dessa etapa do algoritmo podem ser incorporadas na arquitetura distribuída que propomos substituindo a implementação sequencial.

3.2 Implementações Paralelas de Algoritmos de Compressão

Outros trabalhos encontrados na literatura propõem estratégias de paralelização de algoritmos de compressão de dados, para ambientes distribuídos e não-distribuídos. Gilchrist *et al.* [11] e Nascimento *et al.* [18] utilizam o método de compressão de Burrows-Wheeler de forma paralela em ambientes distribuídos (usando bibliotecas MPI) e não-distribuídos (usando memória compartilhada).

Os trabalhos destes autores serviram como referência para a implementação proposta neste trabalho. A diferença principal está na opção que fizemos de construir uma solução voltada para execução em um cluster Hadoop, que permite explorar o princípio de distribuir as tarefas movendo o processamento até onde o dado se encontra, ao invés de trafegar os dados. Além disto, o uso deste *framework* permite o compartilhamento dos recursos computacionais usados com outras aplicações que executam sobre o mesmo ambiente e a adoção das soluções de tolerância a falhas e gerenciamento de recursos já oferecidas por esse ambiente.

3.3 Compressão de dados usando Hadoop

Uma publicação de grande contribuição para este trabalho é a de Urbani *et al.* [27], que apresenta uma solução que utiliza o modelo MapReduce e o *framework* Hadoop para implementar a compressão e a descompressão de arquivos utilizando um algoritmo baseado em dicionário. Ding *et al.* [9] também apresenta um método de compressão implementado utilizando o *framework* Hadoop MapReduce. Estes trabalhos serviram como base para a implementação MapReduce proposta neste trabalho.

Durante o desenvolvimento deste trabalho, nenhum outro trabalho foi encontrado na literatura fazendo uso direto do *framework* Hadoop YARN para compressão de dados. Sendo assim, os resultados obtidos pelas implementações que fazem uso do Hadoop YARN propostas neste trabalho serão comparados apenas com as implementações MapReduce.

4 IMPLEMENTAÇÕES DISTRIBUÍDAS DO ALGORITMO DE HUFFMAN USANDO O HADOOP

Baseado em um estudo inicial de uma aplicação sequencial não-distribuída para comprimir dados utilizando-se o algoritmo de Huffman, foi possível identificar que a maior parte do tempo de execução dela deve-se às leituras e escritas de dados no disco (primeiro e terceiro passos discutidos na seção 2.1.2).

A partir desta análise, ficou evidente que um compressor baseado no algoritmo de Huffman é *I/O bound*, ou seja, os fatores limitantes do desempenho da aplicação são as operações de I/O realizadas na memória secundária. Uma tentativa de melhorar isso é carregar o arquivo inteiro na memória RAM, fazendo com que a aplicação não necessite ler os dados do disco rígido para executar os passos da contagem de caracteres e de codificação do arquivo. Porém, o espaço disponível em uma memória RAM em uma única máquina não é suficiente para armazenar grandes arquivos, fazendo com que apenas uma pequena parte dele possa ser mantido em memória. Com o crescimento do tamanho do arquivo, a porcentagem dos dados do arquivo carregado na memória se torna cada vez menor, fazendo com que este ganho de desempenho seja cada vez menos perceptível. Sendo assim, uma implementação paralela e distribuída pode auxiliar de duas maneiras: (i) permitindo a realização de operações de I/O em disco de modo paralelo; e (ii) aumentando a quantidade de memória RAM disponível, uma vez que cada máquina poderia armazenar um pedaço do arquivo de entrada em sua própria memória.

Neste capítulo, propomos duas alternativas de implementação paralela e distribuída usando o ambiente Hadoop: uma baseada no modelo MapReduce, e outra construída diretamente sobre o framework Hadoop YARN. O desenvolvimento das

soluções baseadas no modelo MapReduce nos permitiu compreender melhor as vantagens e dificuldades desse modelo para a implementação de um algoritmo clássico de compressão de dados, serviu como referência inicial para avaliar alternativas de particionamento e distribuição das principais tarefas de um algoritmo de compressão e para a construção de um modelo de programação específico para compressão de dados usando o framework YARN.

4.1 Implementações Usando Hadoop e MapReduce

Nesta seção, apresentamos as duas propostas de implementações distribuídas usando o modelo MapReduce. Além de servirem como referência inicial, elas serão usadas também na fase de avaliação deste trabalho.

4.1.1 Implementação Inicial MapReduce

Para iniciar o desenvolvimento da implementação MapReduce, foi utilizada como referência a divisão das tarefas explicitadas na seção 2.1.2. A primeira tarefa, que realiza a contagem da frequência de cada byte, foi escolhida para ser executada como um *job* MapReduce devido ao seu modo de funcionamento, que pode ser facilmente representado por uma tarefa deste modelo, de forma bem parecida com o exemplo da contagem de palavras. Para que isto fosse possível, foi necessário implementar um formato de entrada de dados novo, diferente dos formatos fornecidos pelo próprio Hadoop. Esse novo formato foi definido na classe *ByteInputFormat*. Conforme explicitado anteriormente, a classe *TextInputFormat*, nativa do Hadoop, oferece suporte a arquivos de entrada no formato de texto e, juntamente com o seu *BlockRecordReader*, acessa o sistema de arquivos e particiona um arquivo de entrada em pares de chave/valor para serem entregues aos containers de *map*, onde cada valor

corresponde a uma linha do arquivo de texto e a chave é o *offset* deste valor dentro do arquivo inteiro. Para fazer esta divisão, ele se baseia no caracter que faz quebra de linha, separando um arquivo de entrada em linhas. Se este particionador tivesse sido escolhido, a aplicação ficaria restrita à compressão de arquivos de texto.

Como a compressão de Huffman é feita por bytes, o particionador composto pelas classes *ByteInputFormat* e *BlockRecordReader* foi implementado para ler os dados do arquivo de entrada (apenas os que pertencem ao bloco pelo qual o container *map* é responsável) e fornecer um vetor de bytes para cada função de *map* como valor, e, como chave, o *offset* daquele vetor no arquivo total. Para representar o vetor de bytes foi utilizado o tipo *BytesWritable*, e para o *offset*, foi escolhido o tipo *LongWritable*, ambos nativos do Hadoop. Para gerar os pares que serão enviados aos métodos de mapeamento, o particionador tenta alocar um vetor do tamanho do bloco do arquivo de entrada pelo qual ele é responsável. Caso não seja possível, ele tenta alocar um vetor com metade deste tamanho, e este processo é repetido até que seja encontrado o maior espaço de armazenamento encontrado, que será utilizado durante o restante da aplicação. Quando este tamanho é encontrado, ele começa a separar os blocos do arquivo em vários vetores de bytes deste tamanho, e os encapsula em objetos do tipo *BytesWritable*.

Com o particionador definido para o primeiro *job*, o próximo passo é definir o código que as funções de *map* devem executar. Inicialmente, a tarefa associada ao Map foi planejada de forma análoga ao exemplo da contagem de palavras: para cada byte que ela encontrasse em uma chave como entrada, deveria emití-lo como chave e o valor 1, que indicaria uma ocorrência deste byte. Porém, esta abordagem teve um desempenho insatisfatório, devido à maneira como o *framework* opera, escrevendo os dados intermediários em disco. Como a aplicação realiza a codificação de 1 byte, existe um conjunto finito de bytes diferentes em um arquivo, independente do seu tamanho, o que não ocorre no exemplo de contagem de palavras, onde pode existir

um número muito grande e variável de palavras em um arquivo. Desta forma, a alternativa utilizada para melhorar o desempenho da contagem das frequências foi cada container de *map* ter um vetor de frequências, onde cada posição representa a frequência de um determinado byte. Ao final de sua execução, o container emite pares de <*IntWritable , LongWritable*>, onde cada par será composto pelo byte encontrado e sua frequência total naquela parte do arquivo, respectivamente. Para representar o valor, foi escolhido o tipo *LongWritable*, de forma que um container de *map* possa funcionar com blocos grandes de um arquivo sem o risco de ocorrência de *overflow* neste vetor de frequências.

A parte de redução recebe como entrada, obrigatoriamente, a saída das funções de mapeamento (<*IntWritable , LongWritable*>), e realiza a soma dos valores para cada uma das chaves que os containers de *map* emitiram, também no formato <*IntWritable , LongWritable*>. Para que todos os dados fossem escritos em um único arquivo de saída que contivesse a frequência de todos os bytes do arquivo, foi utilizado apenas 1 container de *reduce*.

```
class SymbolCountMap extends Mapper<LongWritable,
   BytesWritable, IntWritable, LongWritable> {
    long[] vetorDeFrequencia[256]

    method map(Object chave, BytesWritable valor Context
       contexto) {
        para-cada (byte b em valor) {
            vetorDeFrequencia[b]++
        }
    }

    method cleanup(Context contexto) {
        para (inteiro i de 0 a 256) {
            se vetorDeFrequencia[i] > 0 {
                contexto.escreve(IntWritable(i) , LongWritable
                   (vetorDeFrequencia[i]))
            }
        }
    }
```

```
}

class SymbolCountReducer extends Reducer<IntWritable,
   LongWritable, IntWritable, LongWritable> {

    method reduce(IntWritable chave, Colecao<LongWritable>
       valores, Context contexto) {
        long soma = 0

        para-cada (LongWritable valor em valores) {
            soma += valor
        }

        contexto.escreve(chave, LongWritable(soma))
    }
}
```

Listagem 4.1: Pseudocódigo para a implementação do *job* MapReduce para a contagem dos símbolos do arquivo

A listagem 4.1 possui o pseudocódigo para a implementação das classes de *Map* e *Reduce* para a implementação da tarefa MapReduce que realiza a contagem dos símbolos em um arquivo.

Encerrado este primeiro *job*, o sistema de arquivos distribuído armazenará um arquivo binário com os pares escritos pelo container de *reduce*. A segunda parte da aplicação, responsável por gerar a árvore de Huffman e as codificações de cada símbolo do arquivo de entrada, a partir de suas frequências, é, então, executada. Como esta parte não faz uso de grandes volumes de dados em memória de acesso lento e, consequentemente, não é muito custosa computacionalmente, ela foi implementada de forma sequencial e não distribuída, além de que não seria trivial adaptá-la ao modelo MapReduce. No Hadoop, todo o código de uma aplicação MapReduce que não segue este modelo é executado em uma máquina virtual Java na própria máquina que iniciou a execução da aplicação, no chamado cliente YARN. Por ser um objeto Java normal, ele utiliza rotinas da API do HDFS para acessar o arquivo com

as frequências e utiliza estes dados para gerar a árvore e a codificação para cada símbolo. Em seguida, escreve-se em um outro arquivo no HDFS os dados sobre esta codificação, também utilizando rotinas fornecidas pelo próprio Hadoop. Isto é necessário pois, neste caso, além do descompressor precisar acessar este arquivo, cada container de mapeamento do segundo *job* MapReduce precisa saber qual codificação utilizar para comprimir o arquivo, e, por limitação do *framework*, isto só pode ser feito através de um arquivo escrito no HDFS.

Em seguida, o segundo *job* MapReduce é disparado de forma distribuída para comprimir, de fato, o arquivo. A primeira implementação deste *job* utilizava as funções de *map* para emitir os códigos de cada símbolo e um único container de redução, responsável por juntar estas codificações em um arquivo de saída. Esta abordagem, porém, teve um desempenho muito ruim, pois uma grande quantidade de dados precisava trafegar entre os nós do cluster, já que este container *reduce* receberia todas as codificações emitidas por todos os outros nós e as juntaria em um único arquivo de saída. Uma alternativa encontrada para otimizar isto foi disparar um container de *reduce* para cada container de mapeamento, fazendo com que todas as chaves emitidas por um container de mapeamento fossem processadas por um único container de redução e, consequentemente, cada container de redução ficou responsável por um bloco do arquivo de entrada, de tamanho definido pelas configurações do sistema de arquivos. Com isto, o *framework* dará preferência a escalonar estes containers em um mesmo nó, diminuindo a quantidade de dados trafegados pela rede.

Como as codificações dos símbolos são compostas por bits e a menor unidade que pode ser transmitida é um byte, foi necessário definir um tipo novo para representar as codificações. Para isto, foi implementada a classe *BytesWritableEncoder* que é composto por um *array* de bytes de tamanho variável e um inteiro que indica a quantidade de bits escritos nele. Um objeto deste tipo pode funcionar como um *buffer*, armazenando os bits de cada uma das codificações concatenando-os. Para

que este objeto possa atuar como um *buffer*, ele possui um método que indica se ele está cheio e outro que deve ser chamado para esvaziá-lo, após seus dados serem escritos no disco, por quem gerencia este *buffer*.

O *framework* executa o método *map* de cada container N vezes, onde N corresponde ao número de pares chave/valor encontrados pelo particionador no bloco do arquivo pelo qual ele é responsável. Em cada uma destas execuções, ele recebe um vetor como valor de entrada, que corresponde aos bytes do arquivo original e instancia um objeto do tipo *BytesWritableEncoder*.

Desta forma, o arquivo de entrada e o particionador utilizados são os mesmos do primeiro *job*, então cada *map* receberá pares com uma sequência de bytes e a posição desta sequência no arquivo. Cada container, antes de começar a processar os pares, deve acessar o arquivo que contém a codificação definida na parte não distribuída da aplicação, carregando esta codificação na memória para que ele possa saber por qual código deverá substituir cada byte da entrada. Esta leitura é feita no método *setup* destes containers. Então, cada container começa a processar os pares recebidos, gerando como valor um objeto com a concatenação das codificações, em bits, correspondente a cada byte lido. Ao final da execução do método *map*, é emitido um par do tipo $<LongWritable\ ,\ BytesWritableEncoder>$. Ao final de sua execução, o container também emite a codificação do símbolo utilizado como marcador de fim de bloco.

Como o *framework* ordena os pares emitidos pelos containers de mapeamento antes de entregá-los aos containers de redução, a aplicação precisa garantir que, nesta ordenação, não seja alterada a ordem dos dados do arquivo, além de garantir que todos os pares de um container *map* sejam processados por um único container *reduce*. Para isto, cada container emite chaves iguais às dos pares processados. Como o particionador intermediário padrão do Hadoop atribuiria estas chaves ordenadas

de forma intercalada, o que embaralharia o arquivo de saída, foi implementado um particionador que atribui chaves contidas em um determinado intervalo a um único container de redução. Para calcular qual container de redução receberá uma determinada chave, ele acessa a configuração do Hadoop para saber quantos bytes cada bloco do arquivo de entrada possui. Por exemplo, se o arquivo de entrada está particionado em blocos de 64 bytes, as chaves entre 0 e 63 (inclusive) serão associadas ao primeiro container de redução, as chaves de 64 a 127 (inclusive) ao segundo, e assim por diante.

Como a codificação é feita a nível de bits, os vetores de bytes encapsulados nos valores emitidos durante o mapeamento nem sempre estarão completos (um vetor com 2 bytes e as codificações armazenadas nele totalizando 10 bits, por exemplo). A tarefa de juntar estes vetores para que eles sejam escritos no arquivo de saída é de responsabilidade dos containers de redução.

Quando todos os containers de *map* encerram sua execução, são disparados os containers de redução, que irão receber, de forma ordenada, as codificações correspondentes a cada uma das execuções do método *map*. Como todas as codificações de uma mesma parte do arquivo de entrada serão processadas por um único container *map* e todos os pares emitidos por ele serão processados por um único container *reduce*, os containers de redução também serão responsáveis apenas por um bloco do arquivo de entrada.

Então, os containers de redução concatenam os bits recebidos, armazenando-os em um objeto do tipo *BytesWritableEncoder*, que funciona como um *buffer*. Quando este *buffer* está cheio e não consegue expandir seu espaço de armazenamento para adicionar novas sequências de bits, o objeto que contém ele (neste caso, o container *map*) deve emití-lo para o *framework* e em seguida esvaziá-lo, para então poder adicionar estes bits. Como este *buffer* é composto por sequências de bits

armazenados em bytes, ele pode não estar com todos os bytes completos, então são escritos no arquivo apenas os bytes completos e, em seguida, ele é esvaziado, restando apenas os bits não completos. Um container, ao terminar de processar todos os pares designados a ele, deverá escrever ainda os bytes que restaram no *buffer*. Isto é feito no método *cleanup* do container, onde ele escreve todos os bytes do *buffer* que contenham algum bit de codificação, mesmo se este byte estiver incompleto. A figura 4.1 mostra como este processo ocorre.

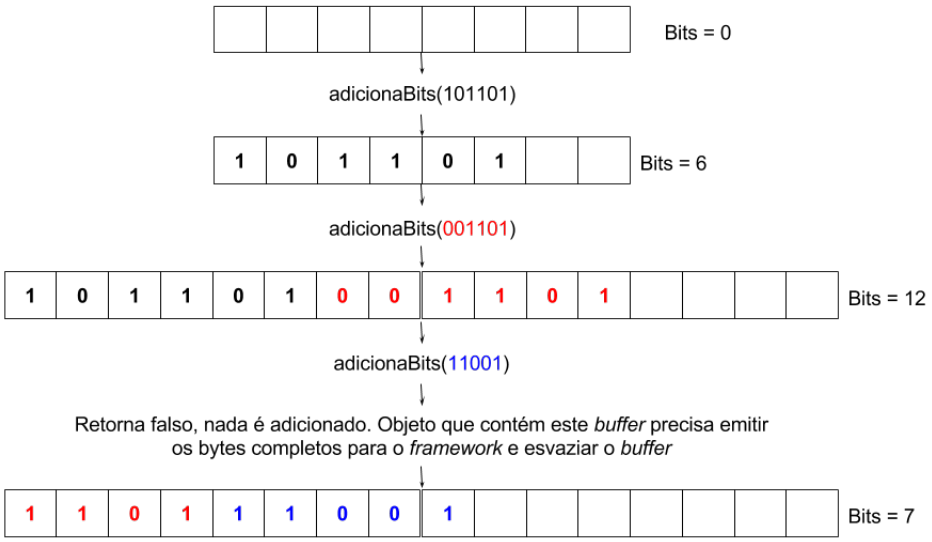

Figura 4.1: Funcionamento de um objeto do tipo *BytesWritableEncoder*

Ao final, cada bloco do arquivo de entrada no sistema de arquivos distribuído terá um arquivo correspondente comprimido, e todos os blocos comprimidos serão escritos em um único diretório, porém podem estar em lugares físicos diferentes (nós diferentes). Como será gerado um arquivo diferente para cada bloco de entrada, cada arquivo pode estar armazenado em um único nó, se seu tamanho for menor do que o limite de cada bloco configurado no HDFS. Caso contrário, ele estará armazenado em mais de uma partição. Para descomprimir o arquivo inteiro, basta percorrer estes arquivos, um a um, seguindo a ordenação dos blocos de entrada, substituindo

as codificações pelo símbolo correspondente no arquivo de codificação até encontrar o marcador de fim de arquivo de uma parte. Ao encontrá-lo, deve-se iniciar a leitura do arquivo seguinte, escrevendo no mesmo arquivo de saída, repetindo-se este processo até que todos os arquivos tenham sido consumidos.

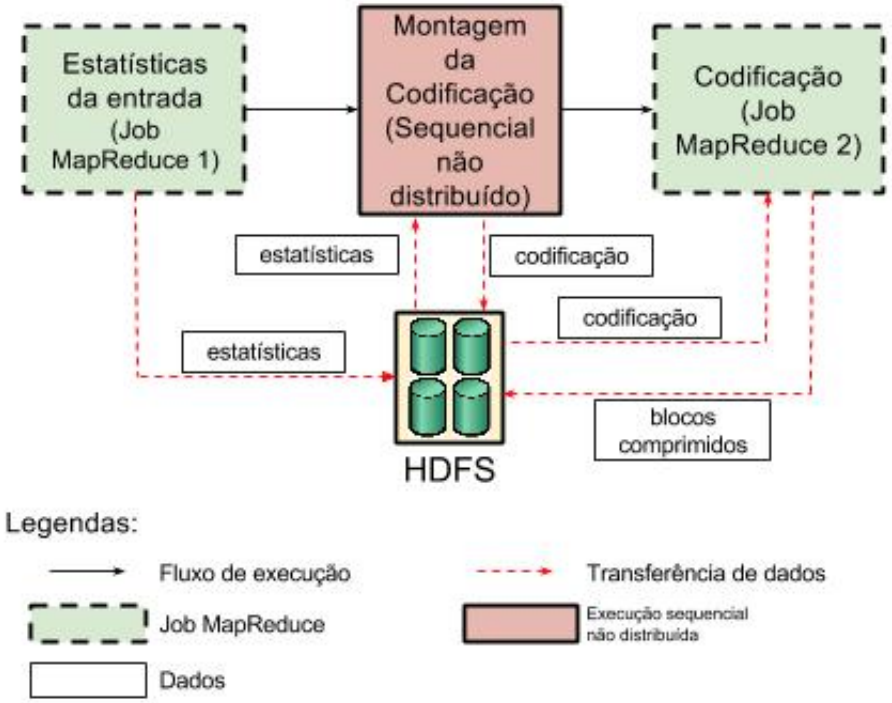

Figura 4.2: Fluxo geral da implementação MapReduce

A figura 4.2 demonstra o fluxo de execução de uma aplicação baseada no modelo MapReduce. O primeiro *job* MapReduce tem como entrada o arquivo original e, como saída, as estatísticas deste arquivo, escrita no HDFS. Em seguida, é executada uma parte sequencial não distribuída para definir a codificação que será utilizada para comprimir o arquivo. Esta codificação é escrita no HDFS para que fique acessível pelos containers do segundo *job* MapReduce, o qual será responsável por codificar os blocos do arquivo original armazenados no HDFS, gerando novos blocos comprimidos.

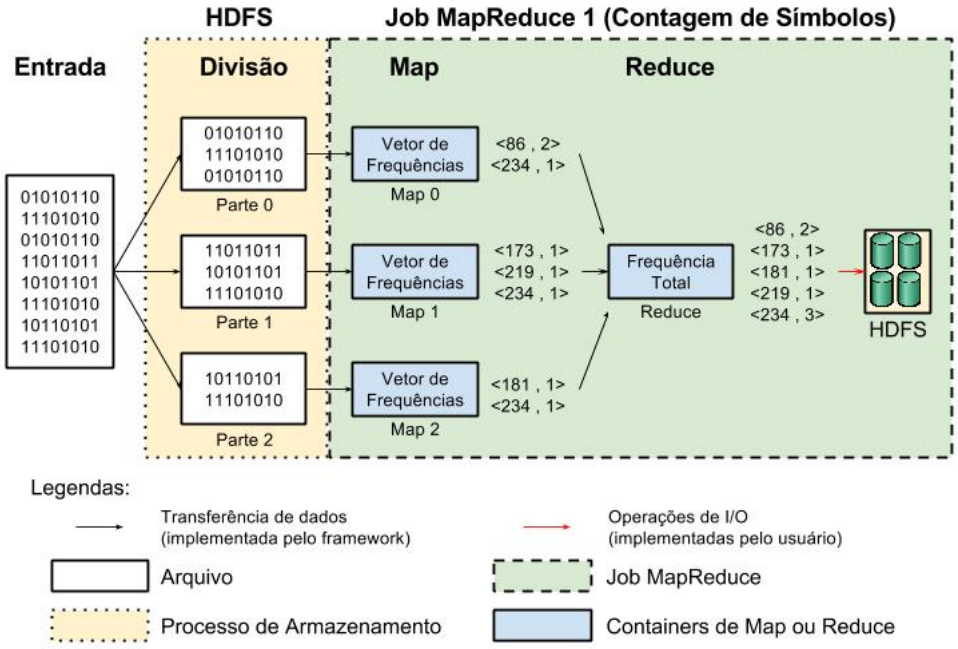

Figura 4.3: Detalhamento do primeiro *job* MapReduce (contagem de símbolos)

Na figura 4.3 podemos observar o funcionamento do primeiro *job* MapReduce, onde serão obtidas as estatísticas do arquivo de entrada. Cada container de *map* possui um vetor de frequência, onde ele vai armazenando as ocorrências de cada um dos caracteres no pedaço do arquivo de entrada pelo qual é responsável. Ao final de sua execução, os caracteres encontrados são enviados como pares de chave/valor ao container de redução, que soma todos eles.

Já no segundo *job* MapReduce representado na figura 4.4, cada um dos containers de mapeamento acessa o arquivo que contém as codificações geradas pelo algoritmo para esta entrada. Cada container de mapeamento codifica a sua parte do arquivo de entrada e envia pares de chave/valor, onde a chave indica qual container de redução irá processá-la e em qual ordem, e o valor é uma sequência de bits codificados.

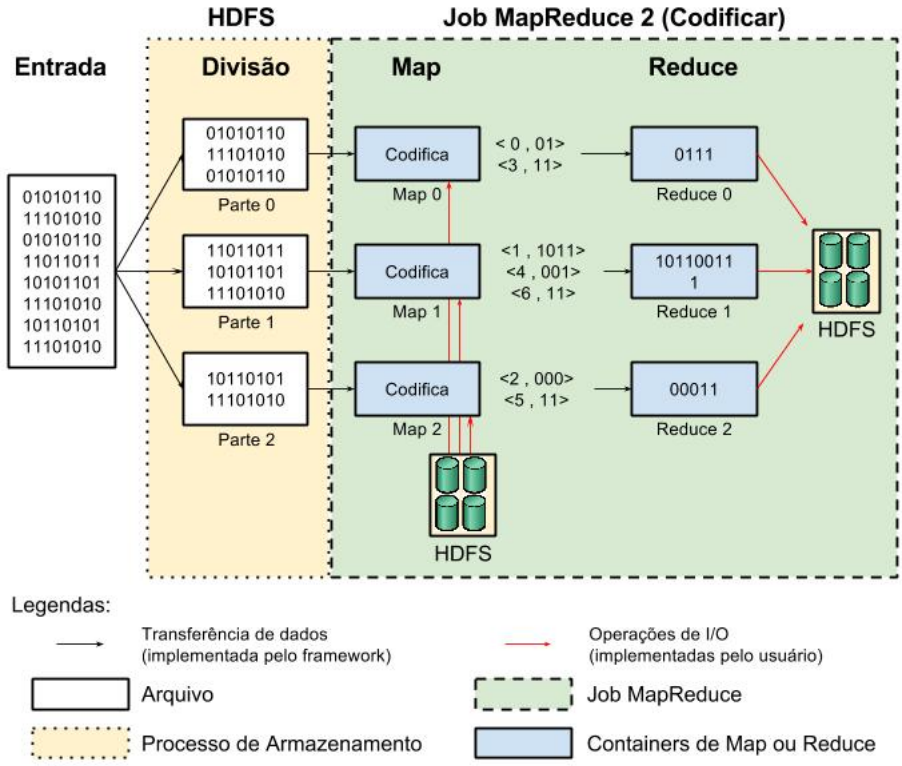

Figura 4.4: Detalhamento do segundo *job* MapReduce (codificação) da implementação inicial MapReduce

4.1.2 Implementação Final MapReduce

Nos testes iniciais da aplicação, a implementação de um compressor baseado no algoritmo de Huffman obteve um desempenho satisfatório (quando comparado com uma implementação sequencial), porém, a partir de um entendimento melhor do modo de funcionamento do *framework*, foram feitas otimizações na aplicação, ainda utilizando o modelo MapReduce.

O Hadoop MapReduce opera escrevendo as saídas dos containers de *map* no sistema de arquivos, para depois acessá-las nos containers *reduce*. Todas estas operações geram uma degradação no desempenho da aplicação, pois este tipo de

acesso ao sistema de arquivos é custoso. No caso do segundo *job* MapReduce da implementação inicial, podemos observar que: (i) existe um mapeamento direto de cada símbolo no arquivo de entrada para uma codificação no arquivo de saída; (ii) este mapeamento é sequencial (o primeiro símbolo no arquivo de entrada corresponde à primeira codificação no arquivo de saída, e assim sucessivamente); e (ii) não há a necessidade de uma função de redução apenas para juntar as codificações, pois isto pode ser feito nos próprios containers de mapeamento. Devido a estes fatos, uma otimização possível é transformar o segundo *job* MapReduce em uma tarefa apenas de mapeamento, não executando containers de redução, de forma que o container de *map* escreva diretamente a saída no arquivo final. Isto diminui a quantidade de informações trocadas através do sistema de arquivos entre os containers, e também evita o *overhead* da alocação de novos containers para o processamento dos valores emitidos pelas tarefas de mapeamento, o que tende a representar uma melhora no desempenho final da aplicação.

Então, para esta nova implementação, cada container de mapeamento do segundo *job* MapReduce passou a ter um objeto do tipo *BytesWritableEncoder* global, que serve como um *buffer*, exatamente igual ao da implementação anterior (figura 4.1. Durante a execução do método *map*, para cada símbolo do arquivo de entrada processado, adiciona-se ao *buffer* a sua codificação. De forma análoga ao container de redução da implementação anterior, quando o *buffer* é completamente preenchido, ele deve ter seus bytes completos emitidos para o *framework* e, em seguida, ser esvaziado, de forma que os bits não escritos sejam mantidos no *buffer*. O *framework*, ao receber estes bytes, irá escrevê-los diretamente em um arquivo de saída, pois não serão executadas as fases de ordenação e nem de redução.

Quando todos os pares já tiverem sido consumidos pelo particionador, o container de mapeamento deverá adicionar ao *buffer* a codificação correspondente ao marcador de fim de arquivo. Em seguida, deverá verificar se sobrou algum bit em

seu *buffer* e, caso exista, emití-lo para o *framework*, que irá escrevê-lo diretamente no disco.

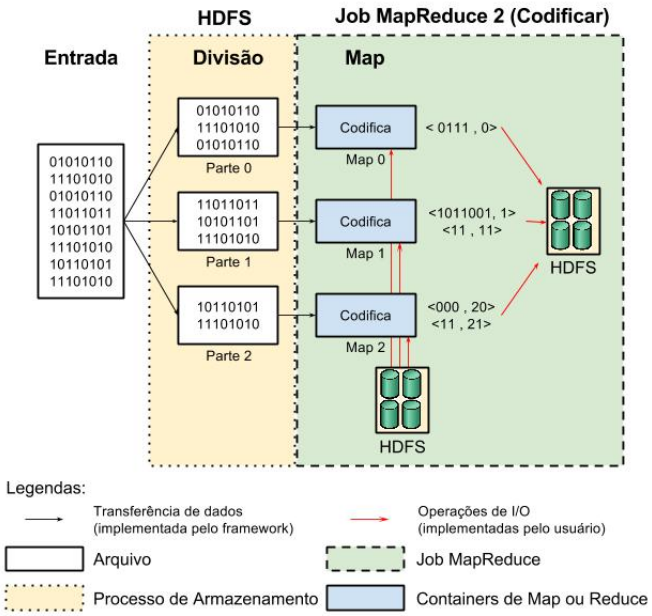

Figura 4.5: Detalhamento do segundo *job* MapReduce (codificação) da implementação final MapReduce.

Esta implementação, portanto, é semelhante à implementação inicial MapReduce do ponto de vista do *framework*, do fluxo de execução e do primeiro *job* MapReduce, podendo ser representados pelas figuras 4.2 e 4.3. A diferença está no modo operacional do segundo *job* MapReduce, representado pela figura 4.5.

4.2 Implementações Usando Hadoop YARN

A implementação proposta para o MapReduce possui algumas restrições devido ao modo operacional do Hadoop MapReduce e, por isso, toda a comunicação entre os containers de *map* e os de *reduce* é feita via operações de leitura e escrita no sistema de arquivos, o que é custoso computacionalmente. Outra limitação é com

relação à gerência de memória, onde um container de *map* não consegue armazenar dados para serem utilizados pelo container *reduce*. Com o objetivo de mitigar estas restrições, visando um aumento no desempenho de algoritmos de compressão de dados que requerem mais de uma leitura do arquivo de entrada, este trabalho propõe uma implementação distribuída para ser executada diretamente sobre o *framework* Hadoop YARN.

Para que uma aplicação use os recursos do Hadoop YARN, ela precisa conter três componentes: (i) *YARN Client*, código sequencial responsável pela comunicação com o *framework* requisitando recursos no cluster para a execução do *Application Master*; (ii) *Application Master* (AM), código responsável por requisitar os containers que a aplicação vai executar e gerenciar o estado deles durante a execução; e (iii) *Container*, código que é executado nos nós escolhidos pelo *Application Master*.

Ao utilizar o Hadoop YARN, o usuário tem a flexibilidade de implementar containers que executem da forma que for mais conveniente. A primeira decisão a ser tomada é com relação à linguagem de programação que será usada para implementar os containers. Como a implementação MapReduce usada no trabalho faz uso da linguagem Java, decidimos continuar usando Java para que as possíveis diferenças nos resultados de desempenho não pudessem ser justificadas pelo linguagem em que cada uma foi implementada.

4.2.1 Primeira Implementação YARN

Inicialmente, a distribuição dos containers desta implementação foi planejada de forma análoga à versão com MapReduce, onde um simples container é disparado para cada bloco do arquivo de entrada armazenado no HDFS, responsável por processá-lo. A diferença, no entanto, é com relação ao modo operacional dos con-

tainers que, no modelo MapReduce, precisavam seguir um fluxo pre-determinado.

Na implementação proposta, os containers carregam os blocos do arquivo pelos quais são responsáveis em seu espaço de memória RAM, e então iniciam a contagem dos símbolos contidos nesses blocos. Um dos containers é escolhido como mestre, e realizará a parte sequencial não-distribuída da montagem da árvore, enquanto os outros containers, os secundários, aguardam o resultado desta etapa para começar a etapa de codificação. Para que o container mestre possa iniciar a montagem da árvore, ele precisa receber a contagem dos símbolos dos outros containers, e para que os containers secundários iniciem a fase de codificação, eles precisam receber a codificação definida pelo mestre. Para isto, foram implementadas rotinas de comunicação entre os containers via *sockets*, para que os containers pudessem se comunicar via troca de mensagens. A escolha por sockets foi feita para explicitar o fato de que a comunicação pode ser feita da maneira que for mais conveniente para a aplicação, usando, por exemplo, bibliotecas próprias para trocas de mensagens.

No YARN, cada container é disparado com um comando executado por um script criado pelo próprio *framework* e a definição deste comando é responsabilidade de quem requisitou este container (cliente YARN ou ApplicationMaster). Este script contém as declarações das variáveis de ambiente do Hadoop YARN e da aplicação, comandos para a configuração do ambiente do container e comandos para disponibilizar localmente no nó os recursos que estão no HDFS. Após a execução de todos estes procedimentos iniciais, é executado o comando para iniciar o container.

Para esta implementação, foi desenvolvido inicialmente o cliente YARN. Esta parte do código é executada de forma sequencial, em uma máquina que tenha capacidade de comunicação com o ResourceManager do cluster YARN. Inicialmente, este cliente recebe como parâmetro de linha de comando o nome do arquivo que será comprimido. Em seguida, ele solicita ao ResourceManager um container para a

execução do ApplicationMaster e disponibiliza no sistema de arquivos distribuído o código executável dele (no formato *jar*). Nesta requisição, ele configura os recursos computacionais que deseja alocar para o ApplicationMaster, o ambiente (variáveis de ambiente na máquina local onde ele será executado), o comando de inicialização da máquina virtual Java que será executada no nó onde o container será alocado e solicita a execução deste comando. Em seguida, ele fica se comunicando com o ResourceManager a cada intervalo de tempo, para saber o status da execução do ApplicationMaster, para descobrir o momento do término da aplicação.

```
{{JAVA_HOME}}/bin/java -Xmx256m br.ufrj.ppgi.huffmanyarn.
   encoder.yarn.ApplicationMaster
   application_1428645096398_0309 /arquivo_a_ser_comprimido
```

Listagem 4.2: Comando para execução do container ApplicationMaster em uma máquina virtual Java

O comando 4.2 mostra um exemplo de comando executado pelo *framework*, em um nó escolhido por ele, para disparar o ApplicationMaster. A variável *JAVA_HOME* é inicializada pelo script de execução, construído automaticamente pelo *framework*, o parâmetro *-Xmx256m* indica que a máquina virtual Java que será executada pode alocar no máximo 256MB de memória RAM e a classe que contém o método *main*, que deverá ser executado é indicada pelo parâmetro *br.ufrj.ppgi.huffmanyarn.encoder.yarn.ApplicationMaster*. Os parâmetros seguintes, *application_1428645096398_0309* e *arquivo_a_ser_comprimido*, são argumentos de linha de comando para o ApplicationMaster, que indicam o identificador daquela aplicação no cluster YARN e o nome do arquivo a ser comprimido. O script de execução, antes de disparar o container, disponibiliza localmente para o nó os recursos especificados pelo ApplicationMaster (neste caso, apenas o arquivo *jar* que contém o código que será executado), que estão armazenados no HDFS. No exemplo, os parâmetros que indicam os arquivos para onde devem ser redirecionadas as saídas *stdout* e *stderr* do ApplicationMaster foram omitidos.

Como o ApplicationMaster não executa nada com relação ao algoritmo de compressão do arquivo, ele apenas gerencia a execução dos containers que farão esta tarefa, o container solicitado para ele não precisa de muitos recursos computacionais, apenas um tamanho de memória razoável para poder armazenar dados sobre os containers e sobre os blocos do arquivo de entrada. No caso, as especificações utilizadas para este container foram de 256MB de memória RAM e 1 núcleo virtual de processamento.

No início de sua execução, o ApplicationMaster instancia objetos relacionados ao ambiente YARN, dentre eles os meios de comunicação com o ResourceManager e com os NodeManagers que irão executar os containers de processamento da aplicação. Em seguida, ele se comunica com o NameNode para obter os metadados do arquivo que será comprimido, que contêm informações sobre os nós onde cada parte do arquivo está armazenada fisicamente. Apesar de todas as partes do arquivo estarem acessíveis de qualquer nó, independente de onde esteja armazenada fisicamente, o *framework* Hadoop YARN permite explorar o princípio de mover a aplicação até onde está o dado, em vez de transferir o dado até onde está a aplicação. Por isto, esta implementação obtém informações do arquivo para dar preferência a executar os containers de compressão nos nós que tenham esta parte do arquivo armazenada localmente.

De posse dos metadados do arquivo, o ApplicationMaster solicita ao ResourceManager a alocação dos containers (um para cada bloco do arquivo), especificando seus recursos computacionais. Como cada container processa apenas um único bloco do arquivo e deve carregá-lo em seu espaço de armazenamento na memória RAM, ele necessita de espaço em memória do tamanho de cada bloco do arquivo (configuração global do HDFS acessível pela aplicação) acrescido de um espaço para as variáveis relacionadas à execução do container. Nos testes iniciais, o valor que permitiu a execução do container de forma correta, para todos os tamanhos do arquivo foi de

256MB (128MB do bloco do arquivo e 128MB das variáveis de execução do container). Com relação ao número de núcleos virtuais de processamento, foi escolhido o valor de 1 núcleo por container. Quando o ApplicationMaster recebe do ResourceManager uma resposta com a alocação de algum container, ele precisa configurar o comando de execução deste container.

São disparados, então, os containers de processamento da aplicação. Todas as informações que um container precisa saber são passadas pelo ApplicationMaster, via argumentos de linha de comando no momento de sua execução. Como explicitado anteriormente, cada container foi planejado para processar um único bloco do arquivo de entrada, então a primeira informação que ele precisa saber são os metadados deste bloco (índice do bloco, *offset* em bytes dele dentro do arquivo original e o tamanho total do bloco). Como os containers se comunicam entre si e um deles (mestre) deve receber as informações, todos os outros precisam saber para qual deles enviar tais informações, então o ApplicationMaster passa como argumento o *hostname* do nó onde o container mestre será executado. Para a escolher um container para ser mestre, foi preciso utilizar um método que permitissse ao ApplicationMaster identificar o mestre e a cada um dos containers saber se ele é mestre ou não. O método utilizado foi escolher como mestre o container responsável pelo processamento do bloco 0 do arquivo. Uma informação importante que o container mestre precisa saber é quantos outros containers estão em execução, pois ele só pode iniciar a fase sequencial após obter os dados de todos eles. Esta informação é passada também pelo ApplicationMaster.

```
{{JAVA_HOME}}/bin/java -Xmx256m br.ufrj.ppgi.huffmanyarn.
   encoder.yarn.Encoder /arquivo_a_ser_comprimido
   2-268435456-134217728 hadoop02 4
```

Listagem 4.3: Comando para execução do container de processamento da implementação

O código 4.3 mostra um exemplo de comando para a execução de um con-

tainer de compressão (com as informações de redirecionamento das saídas *stdout* e *stderr* omitidas, uma vez que estes *streams* possuem apenas informações de logs). De forma análoga ao comando para execução do ApplicationMaster, este comando possui o binário da máquina virtual Java com seus parâmetros (como o tamanho máximo de memória e a classe que contém o método *main*), além dos parâmetros que serão passados para este método. Cabe ressaltar que todos os containers são representados por um único tipo que possui o mesmo código fonte, recebem exatamente os mesmos parâmetros e cada um decide, em tempo de execução, se é mestre ou não, de acordo com o pedaço do arquivo que vai processar e usa esta informação para saber qual trecho de código deverá executar.

Logo no início da execução, um container, seja ele mestre ou não, deve carregar o bloco do arquivo em seu espaço de memória. Feito isto, ele inicia a contagem dos símbolos neste bloco, armazenando-a em um vetor de 256 posições, onde cada uma representa a frequência de cada um dos símbolos. Ao terminar esta parte, o container mestre executa um trecho de código diferente dos secundários. O container mestre abre uma porta de conexão e fica esperando até que todos os outros tenham se conectado nela para enviar seus vetores com as frequências. Já os secundários ficam tentando se conectar nesta porta até conseguirem, e então enviam seus vetores. Tudo ocorre através de rotinas de comunicação bloqueantes, via *sockets* de forma sequencial. Os containers secundários, ao enviarem seus vetores, abrem uma porta de conexão para aguardar os dados da codificação que o container mestre enviará, ficando bloqueados. A comunicação ocorre desta forma mesmo para containers que estejam executando no mesmo nó.

O container mestre faz o somatório de todos os vetores que ele recebeu, monta a árvore de Huffman, gera uma estrutura com a codificação dos símbolos e, em seguida, escreve esta estrutura em um arquivo binário no sistema de arquivos distribuído para que seja possível descomprimir o arquivo. Esta estrutura em sua

memória é serializada em um vetor de bytes para que o mestre possa enviá-la via mensagem aos outros containers que, ao receberem este vetor, farão o processo inverso e estarão prontos para começar a codificar o arquivo. O mestre só poderá codificar o arquivo após terminar de enviar a codificação referente à árvore de Huffman a todos os outros containers. Esta parte é feita diretamente sobre o bloco já carregado na memória, gerando um ganho de desempenho, e cada container vai escrevendo em um arquivo de saída diferente no HDFS. Para isto, cada container tem um *buffer* onde ele armazena os bits correspondentes aos símbolos do arquivo de entrada e, quando este *buffer* estiver cheio, ele é escrito no arquivo e esvaziado (somente os bytes completos), de forma análoga ao *buffer* da implementação final MapReduce. Cada container é responsável por escrever um arquivo de saída no HDFS, o qual corresponde ao bloco que ele está codificando, já comprimido. Estes arquivos são identificados pelo índice do bloco do arquivo de entrada, de forma que eles possam ser ordenados no momento da descompressão (ou seja, o bloco i do arquivo de entrada, tem um arquivo correspondente comprimido, indexado por i, para que a ordem seja mantida ao descomprimir o arquivo)

A figura 4.6 mostra os componentes desta implementação e a maneira como eles se comunicam com o *framework* e entre si, para que seja executada uma compressão distribuída. Cada container é um simples compressor sequencial, responsável por um bloco do arquivo de entrada.

4.2.2 Segunda Implementação YARN

Nos experimentos realizados, a implementação descrita na seção anterior obteve um desempenho bastante satisfatório, superior ao desempenho das implementações MapReduce propostas (detalhamento na seção 5), porém ela não foi capaz de comprimir arquivos muito grandes. Isto ocorre porque existem barreiras de sincroni-

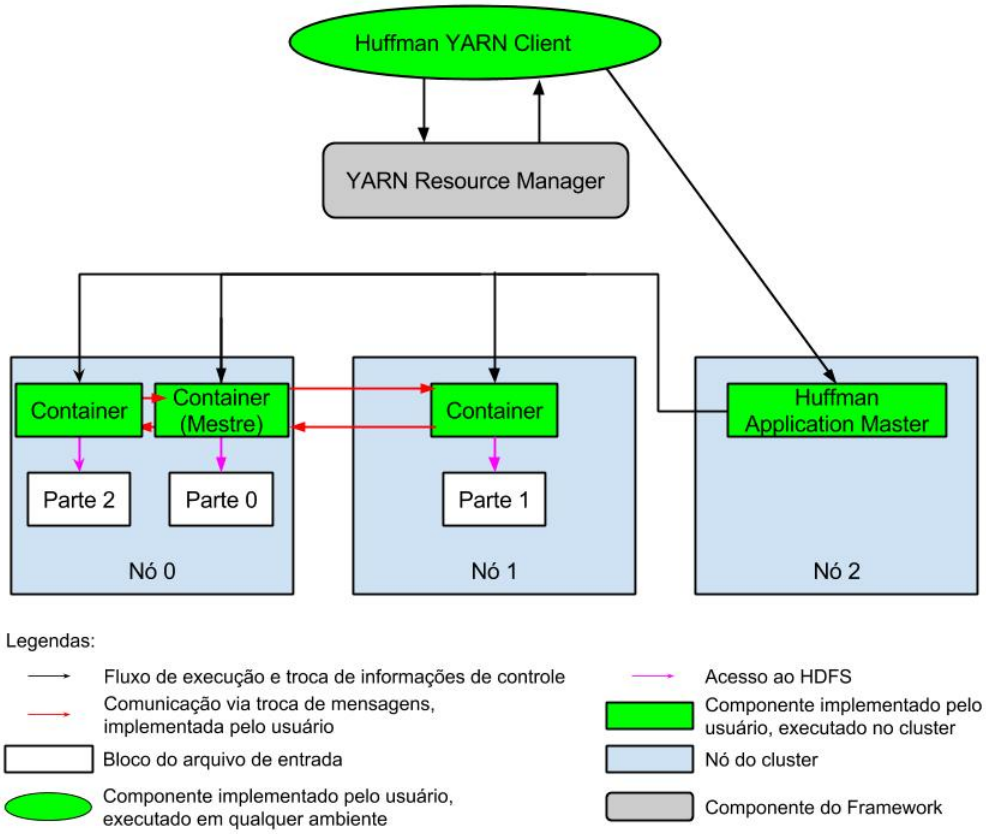

Figura 4.6: Detalhamento da primeira implementação YARN.

zação entre os containers para garantir que o mestre só inicie a montagem da árvore quando todos os blocos do arquivo tiverem suas frequências contabilizadas. Como os containers que estão bloqueados continuam ocupando os recursos do cluster, não é possível alocar outros containers para contabilizar as frequências das outras partes do arquivo, causando uma situação de *deadlock*, pois quando um container é alocado, não há preempção. O tamanho máximo de arquivo que pode ser comprimido com esta implementação fica limitado pelos recursos do cluster (devido à política de alocação de containers), enquanto que as implementações baseadas no Hadoop MapReduce eram limitadas apenas pelo espaço de armazenamento do cluster.

Para contornar este problema, foram feitas alterações, gerando uma segunda

implementação. A implementação inicial YARN foi alterada de forma que não fosse disparado um container para cada bloco do arquivo, mas sim, um único container em cada nó que contivesse algum bloco do arquivo. Cada container, então, seria responsável por processar todos os blocos do arquivo contidos no nó onde ele está sendo executado. Para otimizar o processamento deste container, ele foi implementado de forma paralela usando *multithreading*, onde cada *thread* é responsável por processar um bloco do arquivo. A comunicação entre os containers é executada da mesma forma da implementação inicial e o container escolhido como mestre será o que está responsável pelo processamento do bloco 0 do arquivo de entrada.

Para que esta abordagem fosse possível, o cliente YARN permaneceu igual, mas o ApplicationMaster e os containers de processamento foram alterados. O ApplicationMaster, ao obter todas as informações sobre o particionamento do arquivo de entrada, solicita a alocação de um único container por nó que contenha algum bloco deste arquivo. Nesta requisição, o ApplicationMaster tenta alocar para o container o máximo de recursos que o nó tiver.

Cada container precisa receber do ApplicationMaster uma lista de metadados dos blocos, em vez de um único, como na implementação anterior. Ao ser iniciado, o container carrega os blocos do arquivo em sua memória e, como o container tem uma área de memória limitada, ele carrega nela o máximo de blocos que conseguir armazenar. Na hora de processar estes blocos, ele terá que ler alguns da memória e outros direto do disco. Portanto, quanto menos memória disponível o container tiver, pior será o tempo de execução da compressão, pois ele fará mais acessos ao sistema de arquivos, mas, apesar disto, esta abordagem permite que a aplicação consiga comprimir arquivos de qualquer tamanho, mesmo em ambientes com pouca quantidade de memória RAM disponível.

Ao carregar os blocos do arquivo na memória, o container coloca os metada-

dos deles em duas filas distintas: uma com os que estão localizados no sistema de arquivos e outra com os que foram carregados na memória e inicia a primeira etapa, que realiza a contagem das frequências dos símbolos no arquivo de entrada. Nesta etapa, são disparadas *threads* para a contagem das frequências dos blocos. Uma das *threads* será responsável por processar, exclusivamente, todos os blocos que não foram carregados na memória e todas as outras processarão os blocos diretamente da memória do container. O número ideal de *threads* disparadas é calculado da seguinte forma: uma única *thread* para todos os blocos que serão lidos do sistema de arquivos e uma *thread* para cada bloco carregado na memória. Porém, este número é limitado pela quantidade de núcleos virtuais do container. Sendo assim, se um container responsável por processar 16 blocos, conseguir carregar 10 na memória e tiver 8 núcleos disponíveis, ele iniciará 1 *thread* para os 6 blocos no sistema de arquivos e, ao invés de iniciar 10 *threads* para os arquivos em memória, iniciará apenas 7, totalizando o máximo de 8.

A primeira *thread* iniciada será a responsável por processar os blocos do sistema de arquivos e as demais processarão os dados em memória. Elas farão isto acessando a respectiva fila para pegar o metadado e processar o bloco, lendo-o do sistema de arquivos ou da memória. Quando alguma *thread* terminar de processar um bloco, ela tentará acessar a fila para pegar o bloco seguinte. Então, se a fila estiver vazia, ela encerrará sua execução. Como estas filas são áreas de memória acessadas por diferentes *threads* de forma paralela e envolvem operações de escrita (remoção de um elemento da fila), foram utilizadas estruturas nativas do Java que contivessem rotinas de sincronização, para que não houvesse inconsistência nos dados. A *thread* principal do container fica bloqueada enquanto espera todas as *threads* de processamento se encerrarem.

Cada container tem um único vetor de frequências e os containers secundários deverão enviar este vetor ao container mestre. Como estas frequências são contadas

de forma paralela pelas *threads* durante o processamento dos blocos, teriam que ser implementadas rotinas de sincronização para que todas acessassem uma mesma área de memória sem que houvesse inconsistência nos dados. Neste caso, como alternativa às rotinas de sincronização, foi implementado um vetor de frequências para cada *thread*, onde elas calculavam em sua área de memória as frequências das partes do arquivo pelas quais ela é responsável.

Ao final da execução de todas as *threads* de processamento das frequências em um container, a *thread* principal desse container realiza o somatório dos vetores de frequências das outras *threads* e executa a tarefa de comunicação de forma exatamente igual à implementação anterior. Quando o container mestre tiver recebido os vetores de frequências de todos os containers secundários, sua *thread* principal, de forma sequencial, realizará a etapa de montagem da árvore para gerar as codificações e escreverá um arquivo com estas codificações no HDFS, enquanto as *threads* principais dos containers secundários ficarão esperando a codificação.

Os containers secundários, ao receberem a codificação do mestre, irão iniciar a terceira etapa da compressão, assim como o container mestre, ao enviar a codificação para todos os outros. Para isto, a *thread* principal iniciará a execução de um número de *threads* igual ao que foi disparado para a contagem das frequências. Elas operam da mesma forma, com uma única thread responsável pelo processamento dos blocos no sistema de arquivos e todas as outras processando os que estão carregados da memória, acessando suas respectivas filas para saber qual bloco devem codificar. Neste ponto, não é mais necessário carregar os blocos na memória, pois a única vantagem de mantê-los em memória é para reaproveitar o mesmo bloco na primeira e na terceira etapas da compressão. Cada *thread* acessa a estrutura com as codificações e armazena os bits correspondentes ao símbolo do arquivo de entrada em seu *buffer*. A saída é escrita diretamente no sistema de arquivos, onde cada arquivo comprimido escrito corresponde a um bloco do arquivo de entrada e, ao final, é adicionado o

marcador de fim de arquivo. Assim como na implementação anterior, estes arquivos são identificados pelo índice do bloco do arquivo de entrada, para que seja possível descomprimir de forma ordenada o arquivo.

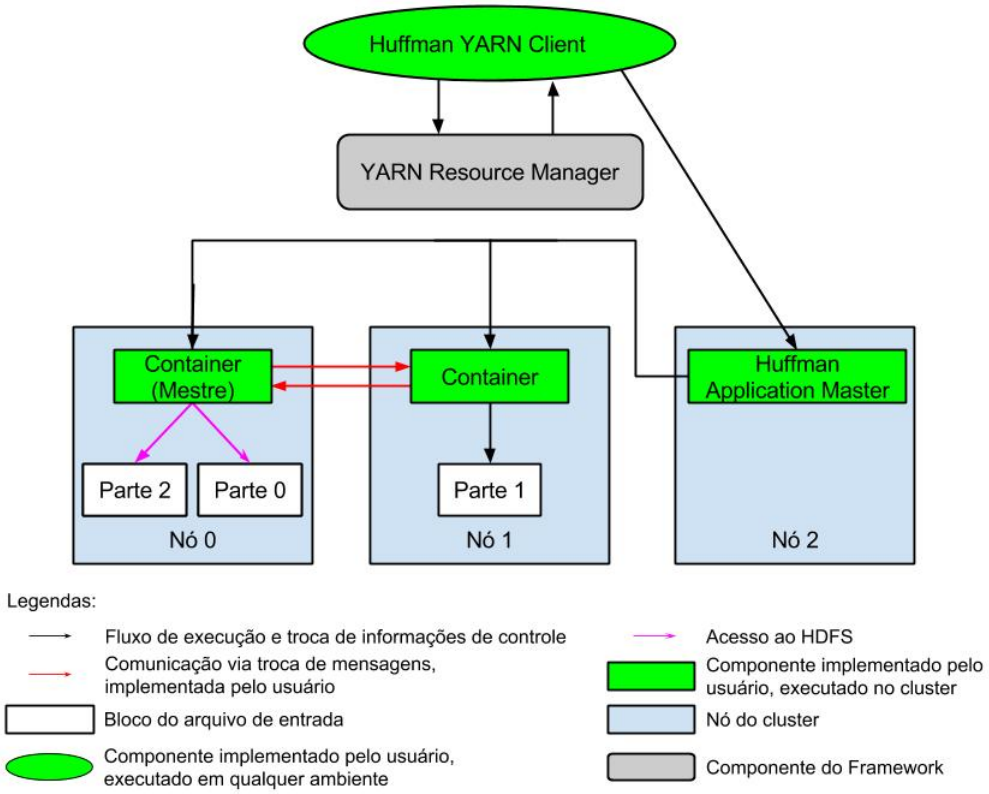

Figura 4.7: Detalhamento da segunda implementação YARN.

A figura 4.7 mostra os componentes desta implementação e a maneira com que eles se comunicam com o *framework* e entre si, para que fosse executada uma compressão distribuída. Cada container, que também foi paralelizado em *threads*, é responsável pelo processamento de mais de um bloco do arquivo de entrada, escrevendo um arquivo de saída correspondente a cada bloco. Apenas uma das *threads* fica responsável pelas operações de leitura de dados do disco, como ilustrado pela figura 4.8.

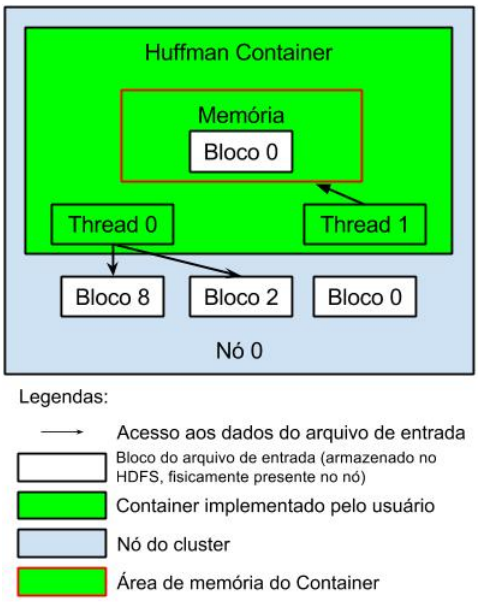

Figura 4.8: Detalhamento da implementação de um container da segunda implementação YARN.

4.2.3 Implementação Final YARN

A segunda implementação YARN proposta teve um desempenho bem próximo da implementação inicial para arquivos de tamanhos pequenos e médios, com uma pequena desvantagem, porém ambas tiveram tempos de execução menores que as implementações MapReduce propostas. Porém, para arquivos grandes, onde a menor parte dos blocos consegue ser carregada na memória, o cenário se inverte e a segunda implementação YARN tem desempenho pior que as implementações MapReduce.

Como nenhuma das duas primeiras implementações YARN desenvolvidas apresentou o resultado esperado (a primeira obteve bom desempenho, porém o tamanho limite do arquivo que pode ser comprimido depende dos recursos de memória RAM do ambiente, e a segunda, apesar de estar limitada apenas pelo espaço de ar-

mazenamento do cluster, não tinha desempenho satisfatório para arquivos grandes), foi realizado um estudo para otimização desta segunda implementação.

Neste estudo, ficou evidente que o baixo desempenho para arquivos grandes era devido ao acesso sequencial ao sistema de arquivos que esta implementação realizava. Então, para otimizá-la, as *threads* foram implementadas para fazerem acessos paralelos ao HDFS, deixando por conta do *framework* a tarefa de otimização destas operações.

Para desenvolver esta implementação, o cliente YARN e o ApplicationMaster utilizados são exatamente iguais aos da segunda implementação YARN proposta, pois a maneira como ela realiza a distribuição das tarefas entre os nós não foi alterada: um container por nó que contém alguma parte do arquivo, cada um responsável por processar e comprimir todos os blocos armazenados fisicamente naquele nó. Todos os containers recebem os mesmos parâmetros.

As alterações efetuadas se restringiram apenas à implementação dos containers de processamento, especificamente na parte de carregar os blocos do arquivo na memória e na forma de operação das *threads* (as que realizam a etapa de contagem de símbolos e as que realizam a etapa de codificação do arquivo).

Um container, ao iniciar sua execução, recebe uma lista com os metadados dos blocos do arquivo a ser processado que estão fisicamente armazenados naquele nó, associa cada um destes metadados à ação de carregá-lo na memória, e enfileira estas associações. Em seguida, ele inicia as *threads* que farão o carregamento do arquivo na memória e processarão os blocos do arquivo. Nesta implementação, todas as *threads* farão acessos paralelos ao disco, deixando a otimização destes acessos por conta do *framework*, então o número de *threads* é calculado baseado na razão de 1 *thread* por bloco do arquivo, limitado pelo número de núcleos de processamento

disponível.

Estas *threads* vão acessando de forma paralela a fila de ações, que contém rotinas de sincronização para que não haja inconsistência nos dados. Ao obter um metadado com a ação de carregar um bloco do arquivo na memória, a *thread* tenta alocar um espaço para esta parte. Caso consiga, ele carrega os bytes deste bloco e, ao terminar, mapeia o metadado dele para o endereço de memória onde está o vetor com estes bytes e adiciona um novo elemento na fila de ações, que corresponde ao metadado desta parte, associado com a ação de processar o bloco. Caso a *thread* não consiga alocar espaço para armazenar este bloco, ela apenas enfileira este bloco com a ação de processar.

Na primeira etapa da compressão, uma *thread*, ao receber uma ação que indica que um bloco deve ser processado, verifica se aquele bloco está associado a um endereço de memória. Se estiver, ela começa a contagem das frequências dos símbolos a partir do vetor de bytes onde está armazenado aquele bloco. Caso não haja nenhum mapeamento daquele bloco para uma área na memória, a *thread* realizará requisições ao HDFS para acessar os dados daquela parte e estas requisições são feitas utilizando um simples *buffer*, pois os acessos em pequenos blocos são mais eficientes que os acessos byte a byte. As *threads* farão esta contagem armazenando os dados em vetores individuais, que serão somados pela *thread* principal, quando todas as outras tiverem encerrado suas execuções, o que ocorre quando a fila de ações estiver vazia.

A comunicação e a etapa de definição da codificação a ser utilizada ocorre de forma exatamente igual à implementação anterior: as *threads* principais dos containers se comunicam com o container mestre (responsável por processar o bloco 0 do arquivo de entrada), os containers secundários enviam seus vetores de frequências para o mestre que faz o somatório total das frequências do arquivo, calcula a co-

dificação a partir da árvore de Huffman montada por ele, escreve a codificação em um arquivo e envia esta codificação para os containers secundários. Ao receber esta informação do mestre, um container inicia a etapa de codificação e o mestre, por sua vez, iniciará quando tiver enviado para todos os containers.

Cada container, antes de iniciar a codificação, adiciona na fila de ações (que está vazia) os metadados dos blocos do arquivo de sua responsabilidade com a ação de processá-los e, então, inicia as *threads* que paralelizarão esta tarefa. O número de *threads* é igual ao da primeira etapa e elas vão acessando os dados da mesma forma: pega na fila os dados de um bloco com a ação de processar, verifica se vai realizar esta tarefa a partir dos dados no sistema de arquivos ou na memória, escreve a codificação no arquivo de saída e adiciona o marcador de fim de arquivo ao terminar. Todos estes acessos ao sistema de arquivos são feitos através de *buffers*, tanto para leitura quanto para escrita dos bytes.

Ao final, cada bloco do arquivo de entrada terá um arquivo de saída comprimido referente a ele no sistema de arquivos distribuído. Para descomprimir o arquivo, basta percorrer todos eles de forma ordenada, escrevendo os símbolos correspondentes às sequências de bits lidos, menos se este símbolo for o marcador de fim de arquivo.

Como, do ponto de vista do *framework*, esta implementação é semelhante à segunda, os componentes desta implementação e a maneira com que eles se comunicam com o *framework* e entre si estão representados na figura 4.7. Cada container, que também foi paralelizado em *threads*, é responsável pelo processamento de mais de um bloco do arquivo de entrada, escrevendo um arquivo de saída correspondente a cada bloco, porém todos eles realizam operações de leitura e escrita do disco simultaneamente, tanto para carregar os blocos do arquivo na memória, quanto para as etapas da compressão, o que os difere da implementação anterior, onde os acessos

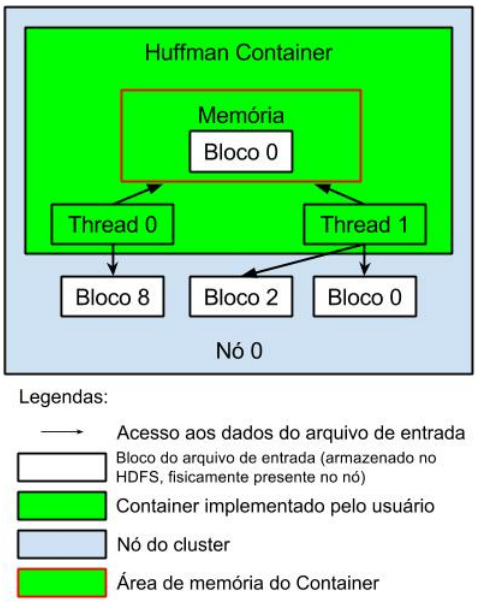

Figura 4.9: Detalhamento da implementação de um container da implementação final YARN.

ao sistema de arquivos paralelo eram feitos de forma sequencial. O detalhamento do funcionamento do container está representado pela figura 4.9.

4.3 Generalização das Implementações Propostas

Os métodos de compressão estatísticos se baseiam em, a partir de estatísticas do arquivo de entrada, definir uma codificação de tamanho variável para os bytes deste arquivo. Esta codificação visa, principalmente, remover a redundância do conjunto de dados presentes no fluxo a ser comprimido. A partir desta codificação, um arquivo de saída novo é produzido, substituindo-se os símbolos de entrada pela codificação definida.

O algoritmo de Huffman é um exemplo de método de compressão estatístico utilizado para compressão de dados sem perda. Em geral, o algoritmo de Huff-

man apresenta boas taxas de compressão, e por isso, apesar de ser uma técnica de compressão desenvolvida há muito tempo, ela continua sendo utilizada até hoje.

A execução de métodos de compressão estatísticos em arquivos grandes é custosa, devido principalmente à grande quantidade de acessos a disco que esses métodos realizam. Quanto maior for o tamanho do arquivo, maior é o tempo de execução das etapas de gerar estatísticas e de codificar o arquivo. Já a etapa de definição de uma codificação depende da variedade do conjunto de entrada, o que indica que seu tempo de execução não depende do tamanho do arquivo de entrada.

Esta característica de processar um mesmo dado duas vezes, em momentos diferentes, é um dos pontos explorados por algumas das implementações propostas, priorizando estratégias que permitem manter o arquivo carregado em memória. Outro ponto explorado é a divisão e distribuição das tarefas de gerar estatísticas e de codificar. Como o *framework* Hadoop armazena os arquivos em seu sistema de arquivos de forma particionada, otimizada para acessos paralelos, e cada nó utilizado para armazenamento também pode ser usado para processamento de dados, as implementações propostas se baseiam no princípio de mover a computação até o dado, diminuindo-se, assim, o custo de transferência de arquivos via rede.

Acreditamos que tais características do *framework* podem ser exploradas por outros algoritmos de compressão baseados em métodos estatísticos. Além disto, o particionamento dos dados já demonstrou ser eficiente para a compressão de dados que se baseiam em outros métodos, como em [11, 18]. Da mesma forma que a implementação proposta explora o recurso de particionar os dados entre tarefas, acessando-os de forma paralela, acreditamos que essa alternativa também poderá ser utilizada com sucesso por outros algoritmos deste tipo.

5 AVALIAÇÃO

Quando se trata de compressão de dados, muitas métricas podem ser aplicadas, podendo ser citadas como exemplo: a complexidade computacional do algoritmo, a memória utilizada durante a compressão ou a descompressão dos dados, o tempo de execução em um determinado ambiente de execução, o quão próxima de sua versão original um arquivo ficará quando for comprimido e descomprimido (aplicável apenas para compressão com perda) e a taxa de compressão obtida pelo algoritmo[21]. Este trabalho trata de uma aplicação completa de compressão, não apenas da parte do algoritmo de compressão em si, e o principal objetivo é propor uma implementação distribuída que facilite a compressão de grandes volumes de dados visando um ganho no desempenho que compense os custos de implementação de uma solução distribuída. Neste capítulo, apresentamos a metodologia adotada para avaliação deste trabalho, o ambiente de experimentação utilizado e os resultados obtidos.

5.1 Metodologia

Para avaliar as implementações propostas, duas métricas principais foram utilizadas: desempenho em termos de tempo de execução e escalabilidade, que são relacionadas ao foco deste trabalho. Para avaliar o ganho de desempenho da solução proposta, foi medido o tempo de execução necessário para comprimir um arquivo variando-se o número de nós de processamento, e para avaliar a escalabilidade foram considerados diferentes tamanhos de arquivos de entrada. Os arquivos já estavam previamente armazenados no sistema de arquivos, então o tempo necessário para o armazenamento e distribuição deles no sistema de arquivos não foram contabilizados.

Como referência inicial, foi utilizada uma aplicação sequencial que implementa o mesmo algoritmo das versões distribuídas. Duas implementações sequenciais foram usadas: uma normal e outra otimizada, onde o arquivo de entrada é mantido na memória visando a diminuição do I/O necessário para realizar a etapa de codificação do arquivo.

Foram utilizados arquivos de diferentes tamanhos como entrada para o compressor. Estes arquivos possuem tamanhos variados (128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB, 16GB, 32GB, 64GB, 128GB, 256GB). Arquivos maiores não foram utilizados devido ao limite de armazenamento do ambiente onde foram executados os testes. Estes tamanhos de arquivos foram escolhidos para que fosse possível experimentar todas as versões propostas, permitindo identificar as vantagens/desvantagens de cada uma delas. Todos são arquivos de texto obtidos de arquivos das bases de dados da Wikipedia, no formato XML, que foram concatenados até obter-se arquivos dos tamanhos desejados. Este tipo de arquivo possui uma grande variadade de caracteres, pois contém caracteres especiais, como parênteses, colchetes e afins. Esta grande variedade é refletida na taxa de compressão, que tende a ser menor, conforme explicado na seção 2.1.2 e também no tempo de execução, que tende a ser maior, devido à grande quantidade de codificações.

Antes de iniciar os testes de desempenho, todas as implementações (sequenciais e distribuídas) foram validadas com relação ao seu funcionamento. Como trata-se de compressão de dados sem perda, um arquivo, após ser comprimido e descomprimido, deve ser exatamente igual ao original antes de ser comprimido. Para esta validação, foram executadas as etapas de compressão e descompressão de cada um dos arquivos, separadamente, por cada uma das implementações, e a comparação do arquivo original com o arquivo descomprimido. O descompressor usado por todas elas foi implementado de maneira sequencial não distribuída.

Para medir o tempo de execução das diferentes implementações propostas, foram feitas pelo menos 3 execuções de cada uma para cada tamanho de arquivo, e ao final foi calculado o tempo médio das execuções. Ao final, foram obtidas as estatísticas sobre o tempo médio de execução de cada versão para cada tamanho de arquivo.

Os experimentos foram realizados em ambientes com 8, 16 e 24 nós de processamento e armazenamento, além de outros 2 para gerência (*Resource Manager* e *Name Node*). Desta forma, os testes de desempenho citados no parágrafo anterior foram realizados para cada uma destas configurações. Para alternar o número de nós de um ambiente para outro, foi necessário desativar a execução do *framework* em alguns nós e recarregar os arquivos no sistema de arquivos.

Espera-se que os tempos para a execução das abordagens distribuídas sejam menores que o tempo da abordagem sequencial para arquivos muito grandes. Com isto, pretende-se evidenciar também o tamanho de arquivo no qual este cenário se inverte, fazendo com que o *overhead* criado pelas versões distribuídas não seja compensado pelo ganho obtido com a paralelização da tarefa.

5.2 Ambiente de experimentação

O cluster Netuno foi utilizado como ambiente de experimentação. Para a execução das versões sequenciais do algoritmo de Huffman não foi necessário nenhum hardware especial, então foi utilizada uma das máquinas do cluster onde foram executadas as versões distribuídas do algoritmo.

5.2.1 Cluster Netuno

Em operação desde março de 2008, o Cluster Netuno é um cluster de alto desempenho que possui 256 nós, armazenamento de arquivos paralelo e uma rede de interconexão de alta qualidade e baixa latência. Atualmente ele está em produção e atende demandas computacionais de larga escala, sendo responsável pela realização dos cálculos de aplicações de diversas instituições de ensino e pesquisa do Brasil. O Projeto Netuno está localizado na Universidade Federal do Rio de Janeiro, mais precisamente no Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais[24].

Para a instalação do cluster Hadoop utilizado nos testes deste trabalho, um *rack* do Netuno foi isolado do restante. Este *rack* possui 32 nós Dell PowerEdge 1950, cada um com 16GB de memória RAM e 2 processadores Intel Xeon 5300 de 4 núcleos cada. Tais nós contam com uma rede de interconexão InfiniBand [19], juntamente com uma rede Gigabit Ethernet, onde cada nó se conecta ao *switch* do *rack* ao qual ele pertence, e as conexões destes *switches* convergem para um principal, através de conexões 10 Gigabit Ethernet.

Em 26 destes nós, foram instalados e configurados um cluster Hadoop com a seguinte organização: um nó será utilizado apenas para a gerência das tarefas de processamento (*Resource Manager*) e para o acesso externo. Outro nó será responsável apenas pela gerência do sistema de arquivos distribuído do Hadoop, o HDFS. Todos os outros nós serão utilizados para o armazenamento (cada nó possui um disco rígido local) e para o processamento dos dados (alguns serão desativados, de acordo com o ambiente que estiver sendo testado). Foi instalada a versão 2.5.1 do Hadoop com suas configurações padrões, sendo as mais importantes o tamanho máximo de cada bloco armazenado no sistema de arquivos distribuído (128MB) e a quantidade de nós que devem armazenar cada um destes blocos (3 nós). Como rede

de interconexão dos nós, foi utilizada a Gigabit Ethernet.

Um dos nós deste *rack* foi utilizado apenas para a execução dos testes das versões sequenciais não distribuídas. Estas implementações utilizaram o sistema de arquivos local da própria máquina, além de não terem sido executadas de forma concorrente com outras aplicações.

5.3 Resultados e discussão

Nos testes de validação do funcionamento da aplicação, todas as implementações propostas, exceto a 4.2.1, foram capazes de comprimir todos os arquivos de forma correta, permitindo que um descompressor gerasse arquivos exatamente iguais ao original ao descomprimí-los. A implementação 4.2.1, como citado anteriormente, era capaz de comprimir arquivos apenas até um determinado tamanho, limitado pelos recursos computacionais do cluster (quantidade de memória RAM nos nós), mas foi utilizada como uma implementação proposta por ser uma alternativa viável para aplicações que não necessitem comprimir arquivos muito grandes ou que sejam executadas em clusters com grandes quantidades de recursos computacionais.

Nos testes realizados, todas as implementações geraram arquivos comprimidos com uma taxa de compressão próxima de 33%. Este valor é muito relativo, pois depende da variedade do conjunto de dados de entrada. Como o foco deste trabalho não é otimizar a taxa de compressão, que depende da natureza do arquivo de entrada, este valor é citado para que possa ser possível prever o tamanho de um arquivo comprimido com alguma destas implementações.

Para executar os testes de validação, os respectivos sistemas de arquivos precisam ter espaço de armazenamento suficiente para armazenar 3 arquivos: (i) o

arquivo original, (ii) o arquivo comprimido e (iii) o arquivo descomprimido. Por ser uma compressão sem perda, o arquivo original e o descomprimido têm exatamente o mesmo tamanho. Já o arquivo comprimido terá aproximadamente 67% deste tamanho. Então, para que fosse possível comprimir um arquivo de 8GB, o sistema de arquivos utilizado deveria ter espaço suficiente para armazenar aproximadamante 21GB.

Sendo assim, nas implementações sequenciais, que não utilizam um sistema de arquivos distribuído, os testes ficaram limitados aos arquivos de no máximo 8GB. Já para as versões distribuídas, este valor varia de acordo com o número de nós utilizados no ambiente, porém deve-se levar em conta que o sistema de arquivos do Hadoop armazena cópias de um mesmo arquivo em mais de um nó. No caso dos ambientes deste trabalho, o fator de replicação escolhido para o sistema de arquivos é 3, o que significa que um arquivo armazenado nele ocupa três vezes o seu tamanho real. Os tamanhos máximos de arquivos comprimidos em cada ambiente distribuído estão representados nos gráficos do tempo de execução x tamanho do arquivo.

Esta seção mostra os gráficos com os resultados obtidos referentes ao tempo médio de execução de cada um dos testes explicitados na seção 5.1, além de uma breve discussão sobre esses resultados.

Nos gráficos a seguir, as legendas *Sequencial* e *Sequencial Otimizada* referem-se às implementações sequenciais não distribuídas do algoritmo de Huffman feitas para este trabalho, onde a otimização refere-se ao carregamento do arquivo de entrada na memória. *MR 1* refere-se à implementação inicial proposta usando o modelo MapReduce da seção 4.1.1 e *MR 2* à implementação final, explicitada na seção 4.1.2. A primeira versão YARN (4.2.1), a segunda versão YARN (4.2.2) e a implementação final YARN (4.2.3) estão com as legendas *YARN 1*, *YARN 2* e *YARN 3*, respectivamente.

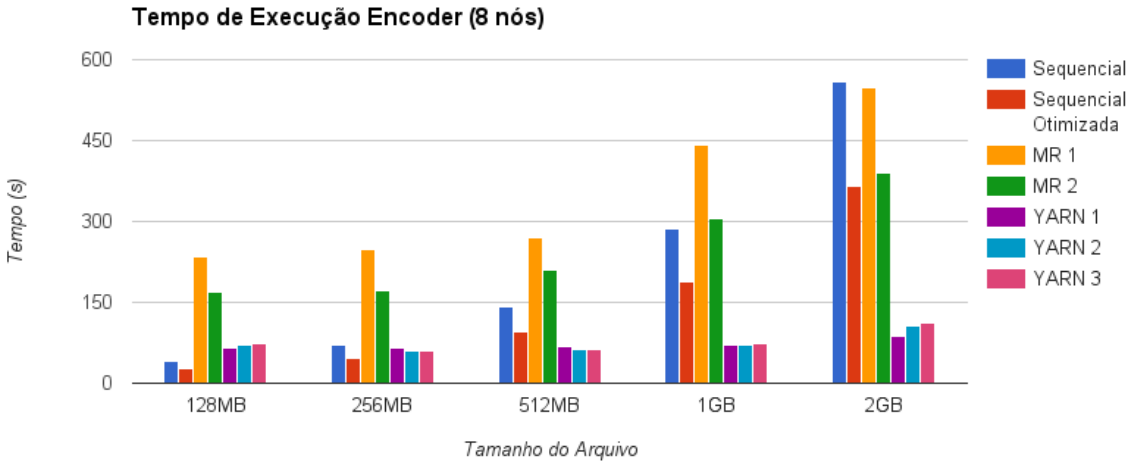

Figura 5.1: Todas as implementações com 8 nós, arquivos menores

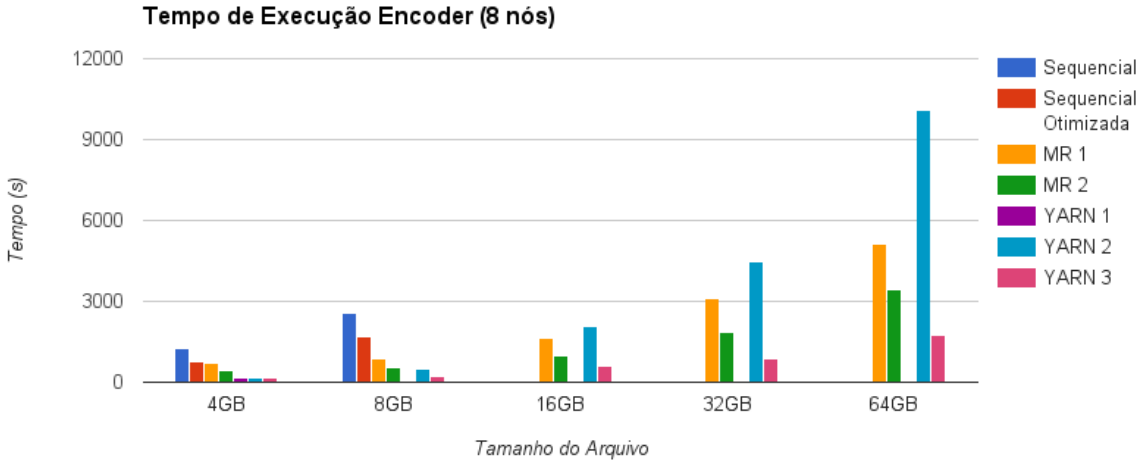

Figura 5.2: Todas as implementações com 8 nós, arquivos maiores

Os gráficos 5.1 e 5.2 mostram um comparativo dos tempos médios de execução de todas as implementações utilizando-se 8 nós de processamento/armazenamento, além da versão sequencial não distribuída, utilizada para efeitos de comparação. Como era esperado, as versões sequenciais tiveram um desempenho melhor ao comprimir arquivos pequenos, devido aos *overheads* de comunicação e distribuição que

existem na computação distribuída, porém é evidente que para arquivos maiores estes custos são compensados pelo processamento paralelo, tornando viável a compressão distribuída para arquivos grandes.

Comparando-se as implementações distribuídas em si, as otimizações da implementação final MapReduce refletem no desempenho, tornando-a melhor que a implementação inicial MapReduce. Já com relação às implementações YARN, a que obteve melhor desempenho foi a primeira, porém, devido ao seu modo operacional, ela ficou limitada a comprimir arquivos até 4GB. É perceptível, também, o baixo desempenho da segunda implementação YARN para arquivos grandes, devido ao fato de ela não realizar acessos paralelos ao sistema de arquivos. Já a implementação final YARN, assim como as implementações MapReduce apresentaram uma diminuição constante no desempenho com o aumento do tamanho do arquivo de entrada.

O tamanho de arquivo onde a implementação *YARN 2* passa a ter um desempenho ruim (comparado com as outras implementações distribuídas) é definido pela quantidade de blocos que a aplicação consegue carregar em sua memória. Os gráficos exibem também como este carregamento na memória pode influenciar no desempenho final da aplicação sequencial.

As implementações distribuídas apresentam um tempo de execução praticamente constante para a compressão de arquivos de 128MB e 256MB, o que ocorre devido ao fato do sistema de arquivos distribuído estar configurado para armazenar cada arquivo em blocos de, no máximo, 128MB. Sendo assim, arquivos menores que este valor possuem apenas 1 único bloco, fazendo com que a aplicação seja executada de forma não distribuída.

Os gráficos 5.3 e 5.4 apresentam os tempos de execução em um ambiente de

16 nós.

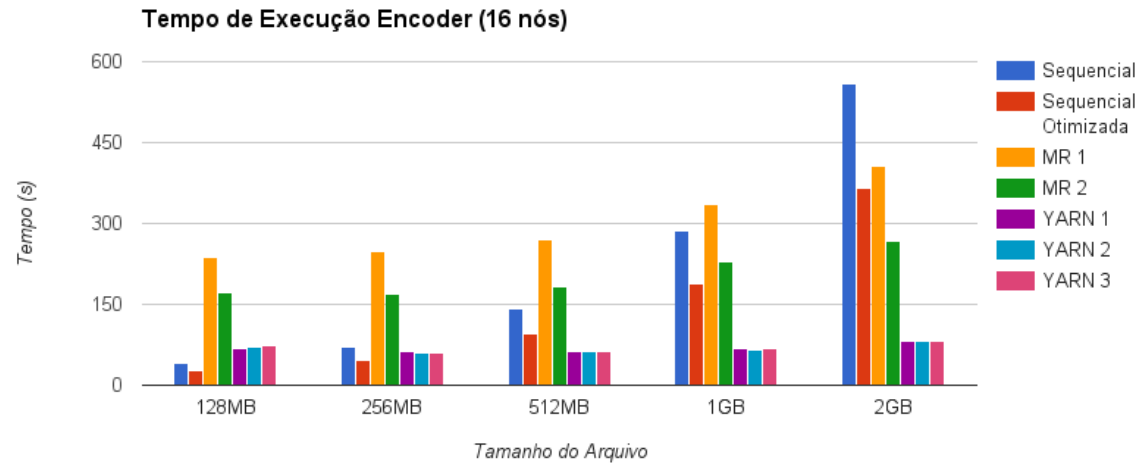

Figura 5.3: Todas as implementações com 16 nós, arquivos menores

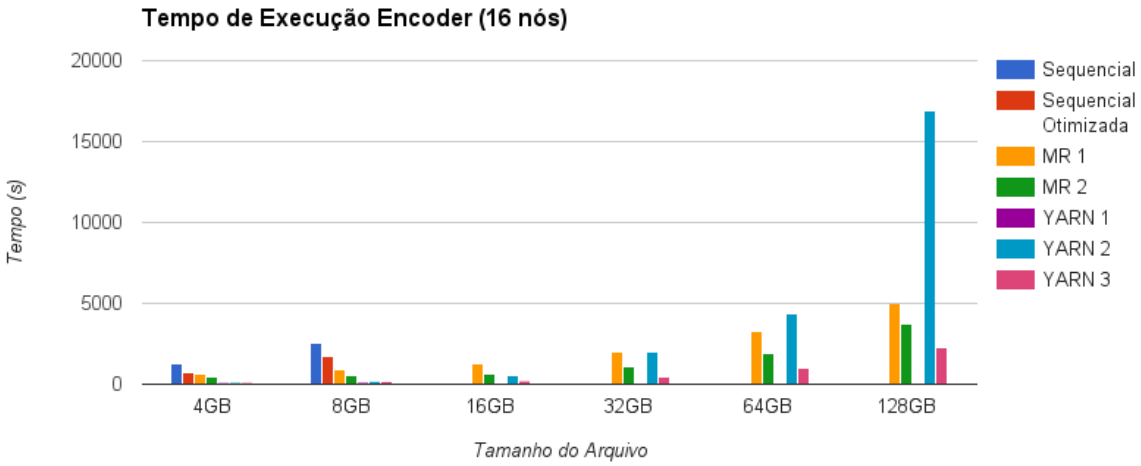

Figura 5.4: Todas as implementações com 16 nós, arquivos maiores

Os mesmos comentários dos gráficos do ambiente de 8 nós podem ser feitos para o ambiente de 16 nós, ressaltando um aumento no tamanho máximo de execução das implementações. Também pode ser observado que a implementação baseada na segunda implementação YARN passa a ter uma degradação no seu desempenho a

partir de arquivos de 16GB e não em 8GB, como ocorreu no ambiente com 8 nós. Isso se deve ao fato de que existem mais nós de armazenamento, então o *framework* distribui os blocos do arquivo de forma menos concentrada. Sendo assim, cada container (que contém a mesma quantidade de memória RAM, independente do ambiente), consegue carregar uma porcentagem maior dos blocos.

Os gráficos 5.5 e 5.6 mostram o comparativo de todas as implementações durante a execução com 24 nós.

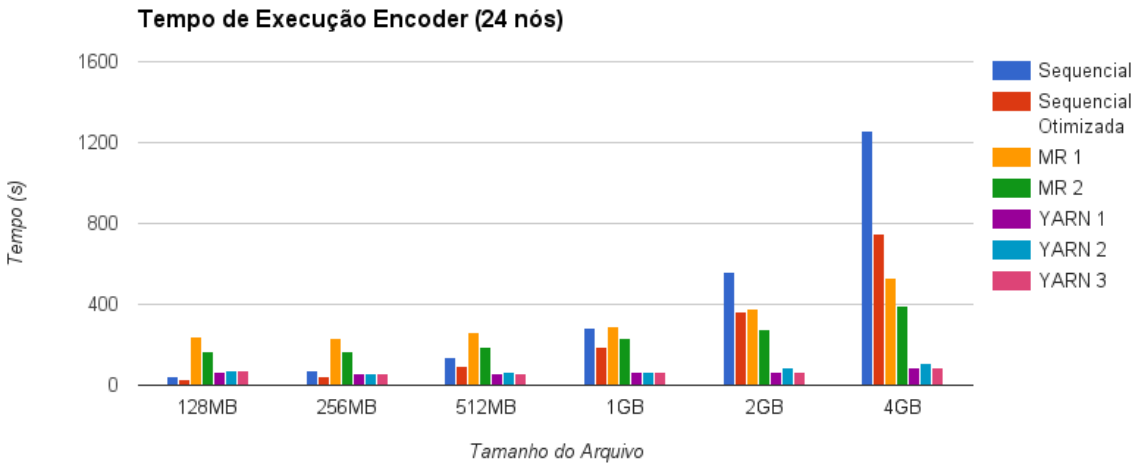

Figura 5.5: Todas as implementações com 24 nós, arquivos pequenos

Baseado nestes resultados e no foco deste trabalho, que é a compressão de grandes volumes de dados, pode-se dizer que a implementação que obteve o melhor desempenho em todos os testes, para arquivos a partir de 512MB, foi a implementação *YARN 3* (seção 4.2.3).

Os gráficos 5.7, 5.8, 5.9, 5.10 e 5.11 mostram comparações do desempenho de cada uma das implementações distribuídas, variando-se o número de nós do ambiente de execução.

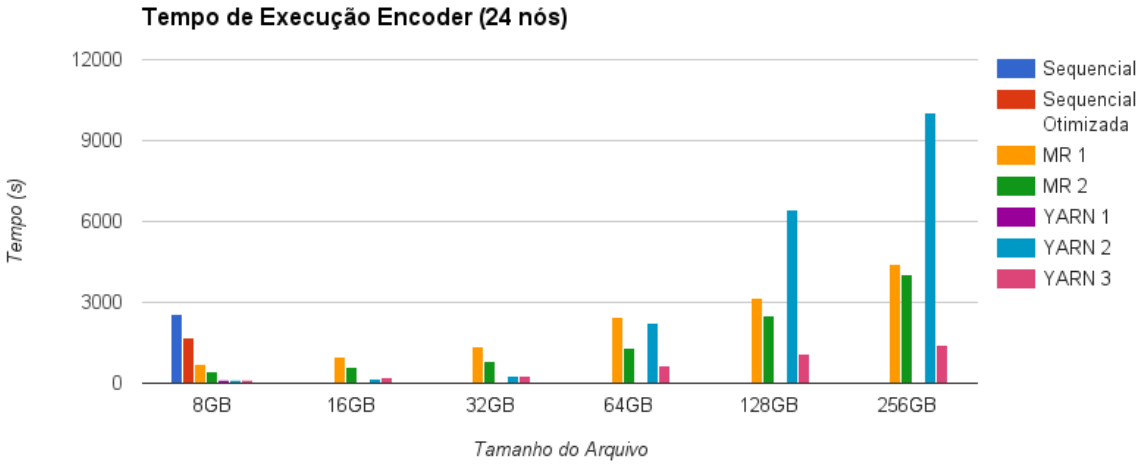

Figura 5.6: Todas as implementações com 24 nós, arquivos grandes

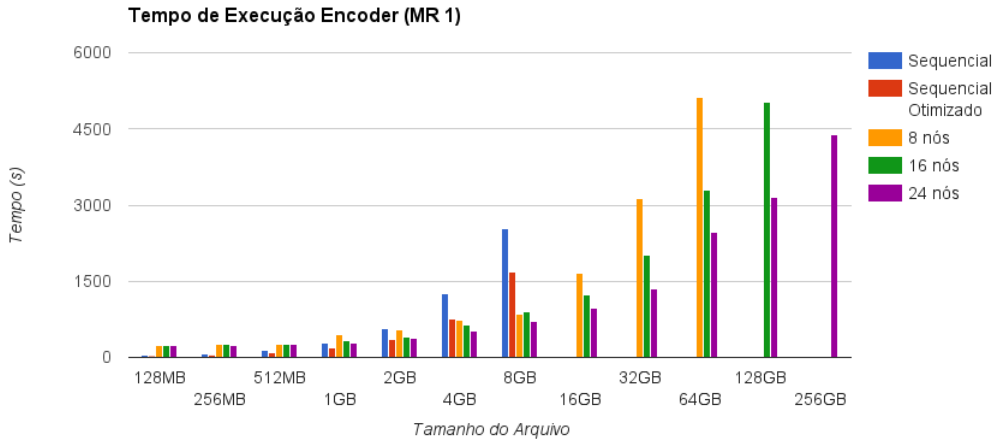

Figura 5.7: Implementação *MR 1* variando-se o número de nós.

A implementação *YARN 1* obteve um desempenho pior com o aumento do número de nós em arquivos de 8GB. Este tipo de resultado ocorre devido ao fato de ela ser implementada executando um container para cada bloco do arquivo de entrada. Esta piora no desempenho pode ser explicada por dois motivos: (i) cada container é executado em uma máquina virtual Java diferente e, consequentemente, um processo diferente em execução. Um grande número de processos passa a ser

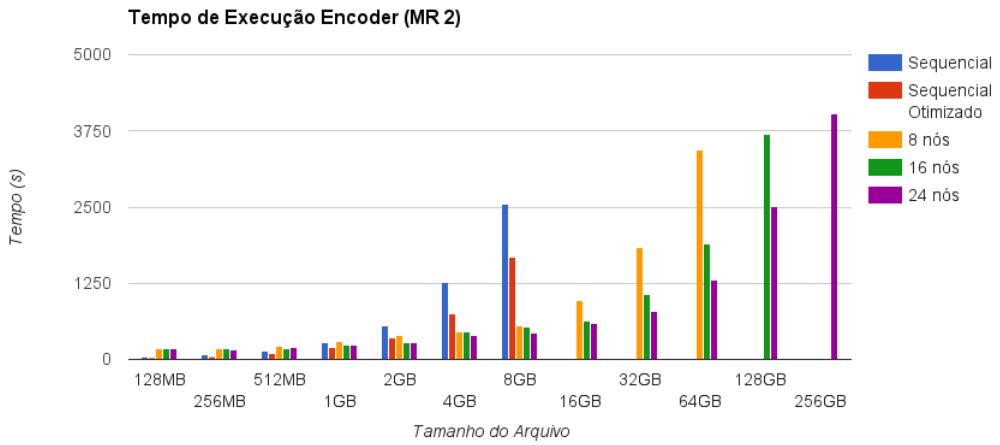

Figura 5.8: Implementação *MR 2* variando-se o número de nós.

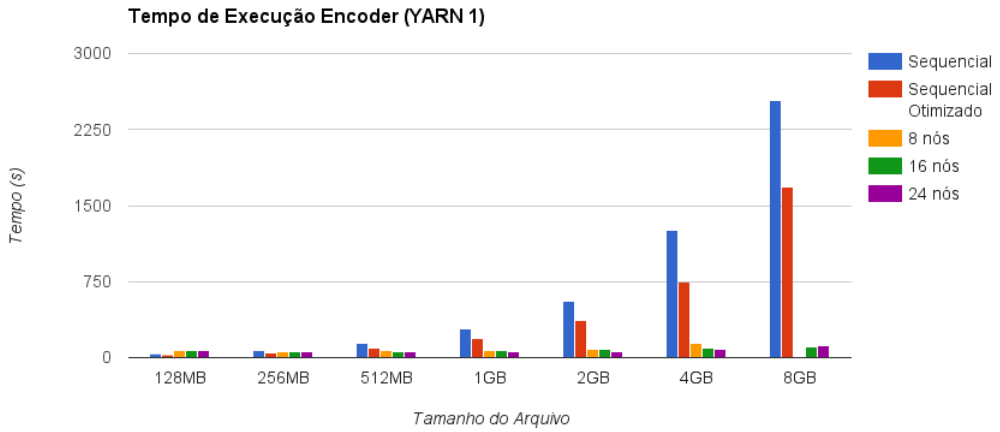

Figura 5.9: Implementação *YARN 1* variando-se o número de nós.

executado de forma concorrente em um mesmo nó, sobrecarregando-o e, consequentemente, (ii) isto faz com que o *framework* aloque os containers em outros nós menos sobrecarregados. Esta alocação dos containers em nós onde os dados não estão armazenados fisicamente gera uma perda de desempenho, pois os blocos precisam ser acessados via rede.

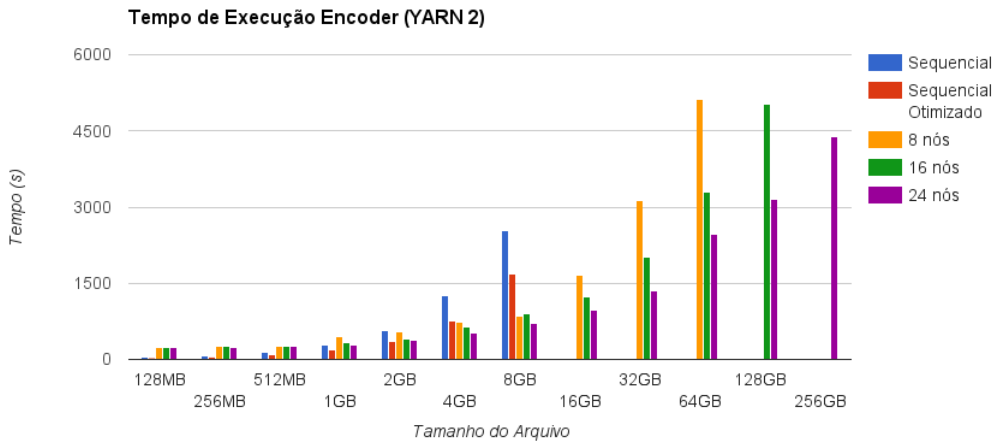

Figura 5.10: Implementação *YARN 2* variando-se o número de nós.

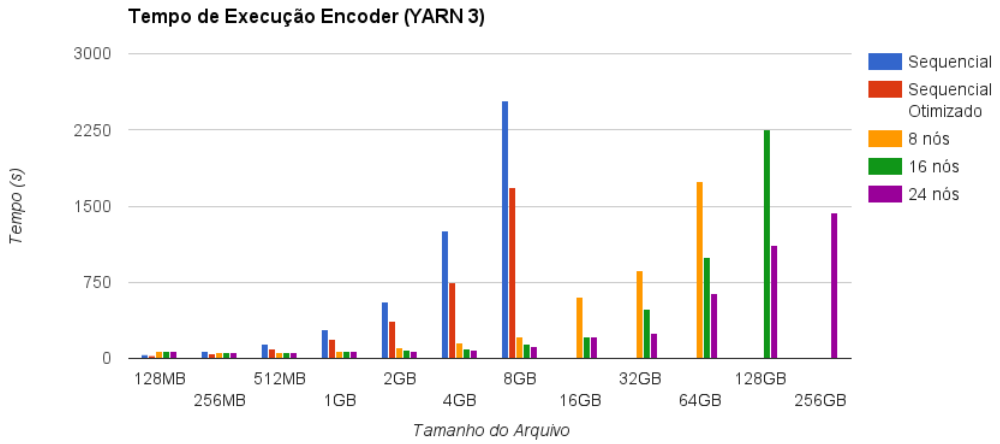

Figura 5.11: Implementação *YARN 3* variando-se o número de nós.

Todas as outras implementações apresentaram uma diminuição no tempo de execução de forma proporcional ao número de nós utilizados. Nas implementações do modelo MapReduce isso já era esperado devido ao modo de alocação dos containers (o *framework* não permite a alocação de muitos containers em um mesmo nó de forma simultânea, evitando uma grande concorrência entre processos em um único nó). Nas implementações *YARN 2* e *YARN 3*, isto ocorre pelo mesmo motivo, já

que existe apenas um único container por nó e cada um executa um número limitado de *threads* simultâneas.

Os gráficos 5.12, 5.13, 5.14, 5.15 e 5.16 apresentam o *speedup* absoluto das implementações ($Tsequencial(1)/T_{paralelo}(p)$), comparadas com a versão sequencial otimizada. Os *speedups* foram calculados para arquivos de tamanhos maiores que 256MB, pois o sistema de arquivos distribuído está configurado para armazenar os arquivos em blocos de 128MB, o que faz com que a compressão de arquivos menores ou iguais a este tamanho seja executada de forma pseudo-distribuída. O tamanho máximo foi o de 8GB, tamanho limite das versões não distribuídas.

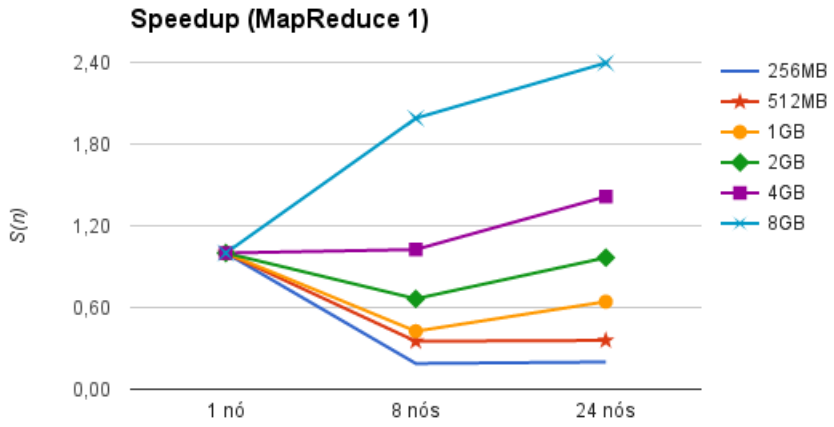

Figura 5.12: *Speedup* absoluto da implementação *MR 1*.

Nos resultados apresentados, é possível perceber que as implementações MapReduce possuem um *speedup* decrescente para arquivos de 256MB a 2GB utilizando de 1 a 8 nós. Isto ocorre pois os custos da distribuição das tarefas não são compensados pelo aumento no desempenho. Já nas implementações YARN, isto ocorre apenas para arquivos de 256MB, devido ao desempenho superior que elas obtiveram.

Os *speedups* de todas as implementações, para arquivos de até 1GB utilizando-

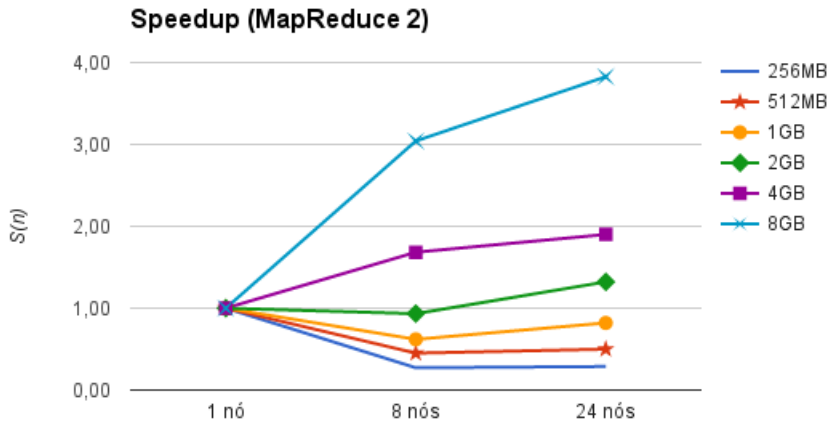

Figura 5.13: *Speedup* absoluto da implementação *MR 2*.

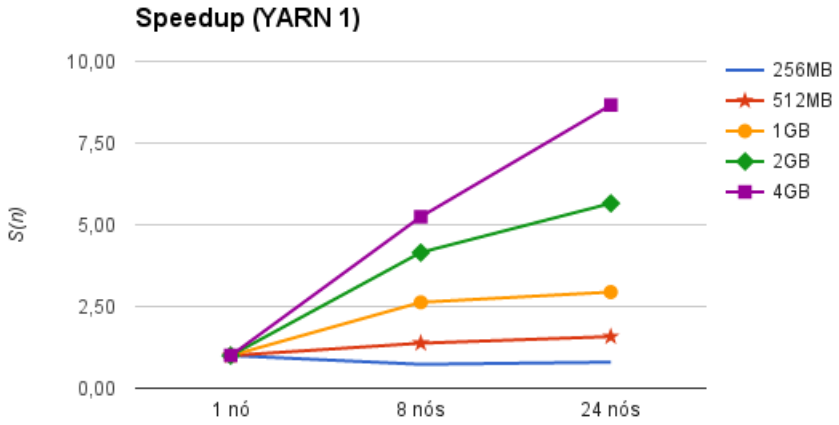

Figura 5.14: *Speedup* absoluto da implementação *YARN 1*.

se de 8 a 24 nós, apresentam valores praticamente constantes. Este fato pode ser justificado pela distribuição entre os nós dos blocos do arquivo de entrada (8 blocos no total), que faz com que todas as implementações sejam executadas em apenas 8 nós, independente do número de nós que o ambiente possui. Porém estes valores apresentam algumas variações, como no caso da implementação *MR 1* que apresen-

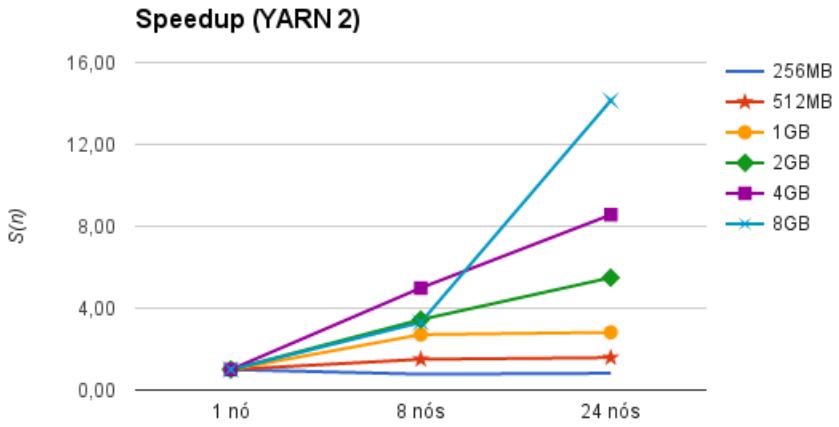

Figura 5.15: *Speedup* absoluto da implementação *YARN 2*.

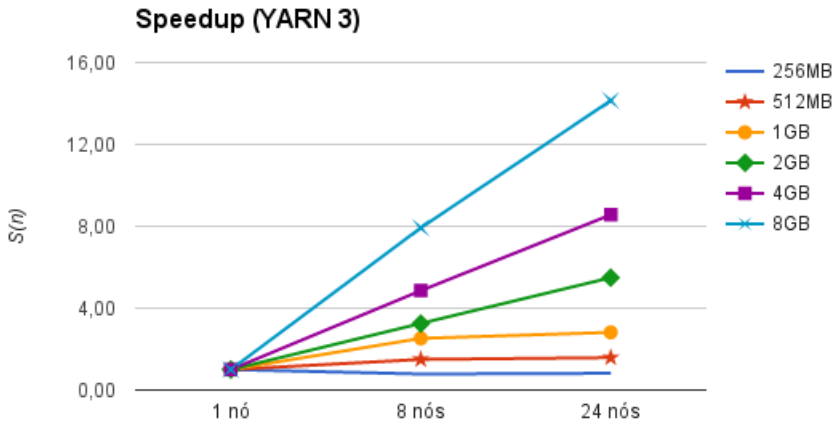

Figura 5.16: *Speedup* absoluto da implementação *YARN 3*.

tou um *speedup* crescente em arquivos de 1GB variando-se o número de nós de 8 a 24. Após um aprofundamento no estudo destes resultados, descobriu-se que isso ocorre devido à heterogeneidade dos discos rígidos do ambiente de execução, que possuem velocidades de acesso diferentes uns dos outros.

Outro fato importante a ser observado sobre os *speedups* é com relação à implementação *YARN 2*, onde as linhas que representam os *speedups* para arquivos de 2GB, 4GB e 8GB se cruzam. Isso se deve ao fato da degradação não linear no desempenho que esta implementação apresenta com o aumento no tamanho do arquivo de entrada, devido aos acessos sequenciais ao sistema de arquivos. Nesta implementação, o *speedup* tende a ir diminuindo com o aumento no tamanho do arquivo de entrada.

Baseado nesta análise dos resultados com relação ao tempo de execução, *speedup* e escalabilidade, a implementação que obteve o desempenho mais satisfatório entre elas foi a implementação final YARN (seção 4.2.3). Além disto, pela sua implementação, é possível que ela seja utilizado para a compressão de arquivos ainda maiores. Tais fatos fizeram com que ela fosse escolhida como a implementação final proposta neste trabalho, porém as outras implementações podem apresentar uma execução viável para algumas aplicações e, portanto, também foram explicitadas aqui.

6 CONCLUSÕES E TRABALHOS FUTUROS

No atual cenário da computação, onde dados massivos têm sido utilizados em larga escala em diversas áreas de conhecimento, a compressão de dados torna-se muito útil para reduzir o custo de armazenamento e o tempo de transmissão que esses dados demandam. A principal motivação para este trabalho foi propor uma implementação distribuída que permita lidar de forma mais adequada com dados dessa natureza, diminuindo o tempo total necessário para a compressão desses dados.

Para isto, propomos algumas implementaçãos distribuídas para o desenvolvimento de aplicações de compressão de grandes volumes de dados, todas elas usando o *framework* Hadoop. Algumas foram projetadas seguindo o modelo MapReduce, utilizando o *framework* Hadoop MapReduce para fazer a comunicação entre a aplicação e o Hadoop YARN, e outras foram desenhadas com uma implementação própria para comunicação com o Hadoop YARN.

Foram propostas algumas implementações de uma aplicação para comprimir arquivos com grandes volumes de dados, baseadas no algoritmo de compressão de Huffman. Estas implementações tiveram seus tempos de execução mensurados para diferentes tamanhos de arquivos, e, para que fosse comparado o tempo de execução destas implementações com uma versão não distribuída, foi implementada também uma versão sequencial do mesmo algoritmo. Este algoritmo foi escolhido devido às características de generalização que ele tem, com relação aos métodos de compressão estatísticos e sem perda.

A partir destas implementações, foram tiradas medidas do desempenho de cada uma delas para diferentes tamanhos de arquivos, cujo tamanho máximo ficou

limitado apenas pelo espaço de armazenamento do ambiente de experimentação (exceto para a primeira implementação YARN proposta). Além disto, foi medido o *speedup* como forma de avaliar o ganho de desempenho e o comportamento de cada uma das implementações com o aumento da quantidade de recursos (nós) do cluster.

Algumas otimizações foram buscadas e avaliadas para cada uma das implementações propostas. A implementação inicial MapReduce foi desenvolvida baseada na associação de uma tarefa para cada bloco do arquivo de entrada (com uma etapa de mapeamento dos dados e outra de redução). Isto ocorre na primeira e na terceira fases da compressão (a segunda fase é realizada de forma sequencial). A otimização desta implementação inicial, proposta na implementação final MapReduce, se baseou no fato de que a tarefa de redução da última fase da compressão poderia ser realizada já no mapeamento, o que geraria um ganho no desempenho devido ao modo operacional do *framework*.

Outa otimização que poderia gerar um ganho de desempenho nas implementações MapReduce seria carregar o arquivo na memória visando diminuir as operações de I/O, uma vez que o compressor precisa ler duas vezes o arquivo de entrada, porém, a implementação desta otimização não foi possível nestas implementações devido ao modo operacional do Hadoop MapReduce. Foi utilizado, então, o *framework* Hadoop YARN, implementando-se uma aplicação que se comunica diretamente com ele. Ao fazer isto, tornou-se possível implementar uma aplicação de maneira distribuída que não ficasse atrelada ao modelo MapReduce.

As implementações YARN propostas se basearam no princípio de mover a computação até o local onde o dado está armazenado (assim como o Hadoop MapReduce) e no princípio de otimizar a aplicação mantendo o máximo possível do arquivo em memória. Para isto, foram inicializados um ou mais containers em cada nó, preferencialmente naqueles que tinham algum bloco do arquivo. Foram imple-

mentados containers *multithread* que fazem acessos paralelos ao sistema de arquivos para as implementações YARN.

Na avaliação dos resultados, as arquteturas inicial MapReduce, final MapReduce e final YARN (seções 4.1.1, 4.1.2 e 4.2.3, respectivamente) funcionaram corretamente para arquivos de tamanhos grandes, ficando limitadas pelo espaço de armazenamento do cluster YARN onde foram executadas. Com relação ao desempenho, todas estas apresentaram boa escalabilidade com o aumento no número de nós e no tamanho da entrada, quando comparadas com a versão não distribuída. Ao avaliar o tempo de execução, a implementação final YARN proposta na seção 4.2.3 apresentou o melhor desempenho dentre todas. É notável o ganho de desempenho utilizando esta implementação, pois com 8 nós ela é capaz de reduzir em cerca de 88% o tempo necessário para a compressão de um arquivo de 8GB de forma não distribuída.

Baseado nestes resultados, a implementação final YARN (seção 4.2.3) foi escolhida como implementação final proposta neste trabalho. Os resultados obtidos demostram também a viabilidade das outras soluções distribuídas e um ganho de desempenho significativo para arquivos acima de 4GB. Além disso, podemos observar que a alternativa de trabalhar diretamente sobre o framework Hadoop YARN (sem utilizar o modelo de programação MapReduce) é promissora para o problema de compressão de dados usando o algoritmo de Huffman.

6.1 Trabalhos Futuros

Pela discussão sobre as características que o algoritmo de Huffman apresenta com relação à sua generalização para outros algoritmos de compressão baseados em métodos estatísticos, é possível que a implementação final proposta neste trabalho

seja, também, generalizada e aplicada para outros algoritmos.

Além disto, é possível, também, estender esta implementação, gerando um *framework* próprio para compressão de dados, análogo ao Hadoop MapReduce. Este *framework* faria a comunicação entre a aplicação do usuário e o Hadoop YARN, permitindo que o usuário especifique apenas como extrair as estatísticas sobre o arquivo de entrada que o método de compressão necessita e como calcular a codificação aplicada. O *framework* permitiria um fluxo de execução distribuído, tolerante a falhas e escalável.

O Apache Spark [30] é um *framework* que, assim como o Hadoop MapReduce, comunica-se diretamente com o Hadoop YARN para executar de forma distribuída as aplicações desenvolvidas utilizando este *framework*. Ele foi criado visando otimizar tarefas iterativas, baseado no fato de utilizar a memória RAM disponível nos nós como um *cache* (distribuído ou não) para os dados do arquivo que a aplicação está processando. Uma outra alternativa de trabalho futuro seria adaptar as implementações propostas neste trabalho para esse *framework* visando explorar as facilidades e benefícios oferecidos por ele (de forma análoga à adaptação feita nas implementações MapReduce).

REFERÊNCIAS

[1] ADDAIR, T. et al. Large-scale seismic signal analysis with Hadoop. **Computers & Geosciences**, [S.l.], v.66, p.145 – 154, 2014.

[2] ATALLAH, M. J. et al. Constructing Trees in Parallel. In: FIRST ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, New York, NY, USA. **Proceedings...** ACM, 1989. p.421–431. (SPAA '89).

[3] AVERY, C. Giraph: large-scale graph processing infrastructure on hadoop. **Proceedings of the Hadoop Summit**, Santa Clara, CA, USA, 2011.

[4] AWS | Amazon Elastic MapReduce (EMR) | Hadoop MapReduce in the Cloud. Acessado em: 2014-10-18, `https://aws.amazon.com/elasticmapreduce/`.

[5] BIG Data | Microsoft Azure. Acessado em: 2014-10-18, `http://azure.microsoft.com/en-us/solutions/big-data/`.

[6] BURROWS, M.; WHEELER, D. J. **A block-sorting lossless data compression algorithm**. Palo Alto, CA 94301: [s.n.], 1994.

[7] DAS, R. et al. Performance and power optimization through data compression in Network-on-Chip architectures. In: HIGH PERFORMANCE COMPUTER ARCHITECTURE, 2008. HPCA 2008. IEEE 14TH INTERNATIONAL SYMPOSIUM ON, Salt Lake City, UT. **Anais...** IEEE, 2008. p.215–225.

[8] DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Commun. ACM**, New York, NY, USA, v.51, n.1, p.107–113, Jan. 2008.

[9] DING, X.; TIAN, B.; LI, Y. A scheme of structured data compression and query on Hadoop platform. In: DIGITAL INFORMATION, NETWORKING, AND WIRELESS COMMUNICATIONS (DINWC), 2015 THIRD INTERNATIONAL CONFERENCE ON, Moscow. **Anais...** IEEE, 2015. p.160–164.

[10] GALLAGER, R. Variations on a theme by Huffman. **Information Theory, IEEE Transactions on**, [S.l.], v.24, n.6, p.668–674, Nov 1978.

[11] GILCHRIST, J.; CUHADAR, A. Parallel Lossless Data Compression Based on the Burrows-Wheeler Transform. In: ADVANCED INFORMATION NETWORKING AND APPLICATIONS, 2007. AINA '07. 21ST INTERNATIONAL CONFERENCE ON, Niagara Falls, ON. **Anais...** IEEE, 2007. p.877–884.

[12] HADOOWIKI - Who Uses Hadoop? Acessado em: 2014-12-10, `http://wiki.apache.org/hadoop/PoweredBy/`.

[13] HASHEMIAN, R. Memory efficient and high-speed search Huffman coding. **Communications, IEEE Transactions on**, [S.l.], v.43, n.10, p.2576–2581, Oct 1995.

[14] HUFFMAN, D. A. et al. A method for the construction of minimum redundancy codes. **proc. IRE**, [S.l.], v.40, n.9, p.1098–1101, 1952.

[15] LIU, X. et al. Implementing WebGIS on Hadoop: a case study of improving small file i/o performance on hdfs. In: CLUSTER COMPUTING AND WORKSHOPS, 2009. CLUSTER '09. IEEE INTERNATIONAL CONFERENCE ON, New Orleans, LA. **Anais...** IEEE, 2009. p.1–8.

[16] MANYIKA, J. et al. **Big Data**: the next frontier for innovation, competition, and productivity. [S.l.]: McKinsey Global Institute, 2011.

[17] MILIDIú, R. L.; LABER, E. S.; PESSOA, A. A. A Work Efficient Parallel Algorithm for Constructing Huffman Codes. In: DATA COMPRESSION CONFERENCE. **Anais...** IEEE Computer Society, 1999. p.277–286.

[18] NASCIMENTO, V. D. do; VERCILLO, D.; SILVA, G. P. da. Um Compressor de Arquivos Paralelo Compatível com o Bzip2. , Campo Grande/MS, Brasil.

[19] PFISTER, G. F. An introduction to the infiniband architecture. **High Performance Mass Storage and Parallel I/O**, [S.l.], v.42, p.617–632, 2001.

[20] SALOMON, D. **Data Compression. The complete reference**. 3rd.ed. [S.l.]: Springer, 2004.

[21] SAYOOD, K. **Introduction to data compression**. [S.l.]: Newnes, 2012.

[22] SCHWARTZ, E. S.; KALLICK, B. Generating a Canonical Prefix Encoding. **Commun. ACM**, New York, NY, USA, v.7, n.3, p.166–169, Mar. 1964.

[23] SHVACHKO, K. V. Apache Hadoop: the scalability update. **login: The Magazine of USENIX**, São Paulo, SP, Brasil, v.36, p.7–13, 2011.

[24] SILVA, V. et al. Arquitetura e avaliacao do cluster de alto desempenho netuno. In: X SIMPOSIO EM SISTEMAS COMPUTACIONAIS, São Paulo, SP, Brasil. **Anais...** WSCAD-SSC, 2009. p.1–10.

[25] SAYOOD, K. (Ed.). **Lossless compression handbook**. [S.l.]: Academic Press, 2002.

[26] TENG, S.-H. The Construction of Huffman-equivalent Prefix Code in NC. **SIGACT News**, New York, NY, USA, v.18, n.4, p.54–61, July 1987.

[27] URBANI, J. et al. Scalable RDF data compression with MapReduce. **Concurrency and Computation: Practice and Experience**, [S.l.], v.25, n.1, p.24–39, 2013.

[28] VAVILAPALLI, V. K. et al. Apache Hadoop YARN: yet another resource negotiator. In: ANNUAL SYMPOSIUM ON CLOUD COMPUTING, 4., New York, NY, USA. **Proceedings...** ACM, 2013. p.5:1–5:16. (SOCC '13).

[29] VITTER, J. S. Design and Analysis of Dynamic Huffman Codes. **J. ACM**, New York, NY, USA, v.34, n.4, p.825–845, Oct. 1987.

[30] ZAHARIA, M. et al. Spark: cluster computing with working sets. In: USENIX CONFERENCE ON HOT TOPICS IN CLOUD COMPUTING, 2., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2010. p.10–10.

APÊNDICE A RESULTADOS

As tabelas A.1 e A.2 mostram os tempos de execução para a compressão de cada tamanho de arquivo, usando cada uma das implementações propostas. O tempo é expresso em segundos e representa o tempo médio das execuções descritas no capítulo Capítulo 5.

Tabela A.1: Tempo de execução para arquivos de 128MB a 4GB

	128MB	256MB	512MB	1GB	2GB	4GB
Sequencial	41	70	141	286	559	1261
Sequencial Otimizada	28	47	95	189	364	752
MR1 (8 nós)	235	249	270	443	548	734
MR1 (16 nós)	237	249	270	336	406	641
MR1 (24 nós)	237	235	265	294	377	532
MR2 (8 nós)	170	173	211	305	391	447
MR2 (16 nós)	172	169	184	229	267	463
MR2 (24 nós)	170	164	190	230	276	396
YARN1 (8 nós)	66	64	69	72	88	143
YARN1 (16 nós)	68	62	64	68	82	90
YARN1 (24 nós)	69	59	60	64	64	87
YARN2 (8 nós)	72	61	63	70	106	151
YARN2 (16 nós)	72	61	62	65	81	107
YARN2 (24 nós)	71	58	63	64	86	109
YARN3 (8 nós)	74	60	64	75	112	155
YARN3 (16 nós)	74	59	63	69	81	98
YARN3 (24 nós)	73	58	60	67	66	88

Tabela A.2: Tempo de execução para arquivos de 8GB a 256GB

	8GB	16GB	32GB	64GB	128GB	256GB
Sequencial	2543	x	x	x	x	x
Sequencial Otimizada	1686	x	x	x	x	x
MR1 (8 nós)	848	1656	3133	5113	x	x
MR1 (16 nós)	898	1242	2012	3297	5039	x
MR1 (24 nós)	704	969	1344	2465	3156	4393
MR2 (8 nós)	554	978	1833	3435	x	x
MR2 (16 nós)	527	630	1072	1893	3693	x
MR2 (24 nós)	441	591	795	1295	2516	4041
YARN1 (8 nós)	x	x	x	x	x	x
YARN1 (16 nós)	112	x	x	x	x	x
YARN1 (24 nós)	120	x	x	x	x	x
YARN2 (8 nós)	511	2055	4484	10071	x	x
YARN2 (16 nós)	185	555	2009	4340	16921	x
YARN2 (24 nós)	116	173	259	2212	6446	10046
YARN3 (8 nós)	213	610	861	1747	x	x
YARN3 (16 nós)	148	217	491	993	2251	x
YARN3 (24 nós)	119	215	246	642	1117	1439