Capítulo 3

Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades

Alfredo Goldman, Fabio Kon, Francisco Pereira Junior,
Ivanilton Polato, Rosangela de Fátima Pereira

Abstract

Advancements on the Internet popularity over the last decade and the increase in volume and complexity of services available on the Web led to the generation of massive amounts of data. To process these data, both performance and availability are critical factors that need to be evaluated since conventional data management mechanisms may not provide adequate support. One existing solution is the Apache Hadoop, a framework for storage and processing data at large-scale. Hadoop offers as main tools MapReduce, responsible for distributed processing, and the Hadoop Distributed File System (HDFS), for storing large data sets in a distributed way. Apache Hadoop has been considered an effective tool and is currently used by large corporations such as IBM, Oracle, Facebook, and Yahoo! among others. This course will introduce the main concepts of the Apache Hadoop framework, demonstrating its features, advantages, applications, components, and a study of new trends on this tool.

Resumo

O grande avanço na popularização da Internet na última década bem como o aumento na quantidade e complexidade dos serviços oferecidos na Web levou à geração de quantidades massivas de dados. Para o processamento desses dados, desempenho e disponibilidade são fatores críticos que precisam ser avaliados, pois mecanismos convencionais de gerenciamento de dados não oferecem o suporte adequado. Uma solução proposta é o Apache Hadoop, um arcabouço para o armazenamento e processamento de dados em larga escala. O Hadoop oferece como ferramentas principais o MapReduce, responsável pelo processamento distribuído, e o Hadoop Distributed File System (HDFS), para armazenamento de grandes conjuntos de dados, também de forma distribuída. Embora recente, o Apache Hadoop tem sido considerado uma ferramenta eficaz, sendo utilizado por grandes corporações como IBM, Oracle, Facebook, Yahoo! entre outras. Este curso apresentará os conceitos principais do Apache Hadoop, demonstrando também suas características, vantagens, aplicações, componentes, bem como um estudo das novas tendências em torno dessa ferramenta.

3.1. Computação Paralela, Big Data e Hadoop

Um dos grandes desafios computacionais da atualidade é armazenar, manipular e analisar, de forma inteligente, a grande quantidade de dados existente. Sistemas

corporativos, serviços e sistemas Web, mídias sociais, entre outros, produzem juntos um volume impressionante de dados, alcançando a dimensão de *petabytes* diários (White, 2010). Existe uma riqueza muito grande de informações nesses dados e muitas empresas não sabem como obter valor a partir deles. A maioria desses dados é armazenada em uma forma não-estruturada e em sistemas, linguagens e formatos bem diferentes e, em muitos casos, incompatíveis entre si.

Esses gigantescos conjuntos de dados têm se tornando uma valiosa fonte de informação. Isso porque, as detentoras desses dados passam a ser avaliadas não somente pela inovação de suas aplicações, mas também pelos dados que elas mantêm e principalmente pela potencialidade que eles podem por ventura trazer. A empresa Google não possui um alto valor agregado somente por seu poderoso algoritmo de busca de páginas Web e seus inúmeros serviços disponíveis, mas também por manter um grande volume de dados oriundos de seus usuários. São esses dados que, ao passarem por análises, tendem a se tornar valiosos, permitindo a criação de soluções inteligentes e bem direcionadas aos seus usuários.

Outro exemplo no qual pode-se constatar um enorme volume de dados é no Facebook. Inicialmente, o Facebook utilizava em sua infraestrutura um banco de dados relacional para o armazenamento dos seus dados. Porém, em uma rápida expansão, a empresa passou de uma quantia de 15 *terabytes* em 2007, para 700 *terabytes* em 2010, tornando essa infraestrutura inadequada (Thusoo, et al., 2010). Em 2012, com cerca de 900 milhões de usuários ativos, e mantendo informações analíticas desses usuários, esse número já ultrapassou os *petabytes*. Hoje, o Facebook é considerado a maior rede social do mundo e, também, uma das empresas mais valiosas.

Baseado nessas e em outras aplicações que possuem um volume gigantesco de dados, surgiu o conceito denominado "Big Data" (IBM, 2011). Esse termo faz menção não apenas ao volume, mas também à sua variedade e a velocidade necessária para o seu processamento. Como existem diversos recursos geradores para esses dados, surge uma enorme variedade de formatos, alguns estruturados e outros não estruturados. Para geradores de dados estruturados, temos como exemplos sistemas corporativos e aplicações Web; já para os não-estruturados, temos os *logs* desses sistemas, as páginas Web, as mídias sociais, celulares, imagens, vídeos, sensores, microfones. Por fim, "Big Data" também está relacionada à velocidade, pois muitas das novas aplicações precisam de respostas em um curto prazo e, em muitos casos, em tempo real.

As aplicações "Big Data" fazem da computação o mecanismo para criar soluções capazes de analisar grandes bases de dados, processar seus pesados cálculos, identificar comportamentos e disponibilizar serviços especializados em seus domínios, porém, quase sempre esbarram no poder computacional das máquinas atuais. Os ditos problemas grandes ou complexos chegam a consumir horas ou dias de processamento nas arquiteturas convencionais. Embora em constante evolução, os recursos computacionais convencionais são insuficientes para acompanhar a crescente complexidade das novas aplicações.

Impulsionada pela grande demanda de computação e por restrições físicas das arquiteturas convencionais, a computação paralela e distribuída acena como alternativa para amenizar alguns dos grandes desafios computacionais. Esse modelo de computação possui atualmente um papel fundamental no processamento e na extração de informação relevante das aplicações "Big Data". Essa computação é normalmente realizada em

aglomerados (*clusters*) e grades computacionais, que com um conjunto de computadores comuns, conseguem agregar alto poder de processamento a um custo associado relativamente baixo.

Muito embora a computação paralela e distribuída seja um mecanismo promissor para apoiar a manipulação das aplicações "Big Data", algumas de suas características inibem sua utilização por novos usuários. Dividir uma tarefa em subtarefas e então executá-las paralelamente em diversas unidades de processamento não é algo trivial. Inclusive, se o tamanho e a divisão das subtarefas não forem bem dimensionados, isso pode comprometer totalmente o desempenho da aplicação. Além disso, o programador precisa: extrair a dependência entre os dados da aplicação; determinar um algoritmo de balanceamento de carga e de escalonamento para as tarefas, para garantir a eficiência do uso dos recursos computacionais; e, garantir a recuperação ou a não interrupção da execução da aplicação caso uma máquina falhe.

Foi nesse contexto que foi desenvolvido o Apache Hadoop[1], um arcabouço (*framework*) para o processamento de grandes quantidades de dados em aglomerados e grades computacionais. A ideia de promover soluções para os desafios dos sistemas distribuídos em um único arcabouço é o ponto central do projeto Hadoop. Nesse arcabouço, problemas como integridade dos dados, disponibilidade dos nós, escalabilidade da aplicação e recuperação de falhas ocorrem de forma transparente ao usuário. Além disto, seu modelo de programação e sistema de armazenamento dos dados promovem um rápido processamento, muito superior às outras tecnologias similares. Atualmente, além de estar consolidado no mundo empresarial, o arcabouço Apache Hadoop também tem obtido crescente apoio da comunidade acadêmica, proporcionando, assim, estudos científicos e práticos.

3.1.1. Histórico

Desde sua origem até hoje, o Hadoop apresentou uma rápida evolução. A seguir apresentamos alguns dos fatos marcantes que nortearam a evolução e a história do Hadoop, como é hoje conhecido.

- Fevereiro de 2003: a Google buscava aperfeiçoar seu serviço de busca de páginas Web - ferramenta pela qual ficou mundialmente conhecida - almejando criar uma melhor técnica para processar e analisar, regularmente, seu imenso conjunto de dados da Web. Com esse objetivo, Jeffrey Dean e Sanjay Ghemawat, dois funcionários da própria Google, desenvolveram a tecnologia MapReduce, que possibilitou otimizar a indexação e catalogação dos dados sobre as páginas Web e suas ligações. O MapReduce permite dividir um grande problema em vários pedaços e distribuí-los em diversos computadores. Essa técnica deixou o sistema de busca da Google mais rápido mesmo sendo executado em computadores convencionais e menos confiáveis, diminuindo assim os custos ligados a infraestrutura.
- Outubro de 2003: depois do desenvolvimento da tecnologia MapReduce pela Google, três de seus funcionários: Sanjay Ghemawat, Howard Gobioff, e Shun-Tak Leung, publicaram um artigo intitulado *The Google File System*

[1] http://hadoop.apache.org

(Ghemawat, Gobioff, & Leung, 2003). Esse trabalho descreve um sistema de arquivos implementado por eles, chamado Google File System (GFS ou ainda GoogleFS). O GFS é um sistema de arquivos distribuído que foi criado para dar suporte ao armazenamento e o processamento do grande volume de dados da Google por meio da tecnologia MapReduce.

- Dezembro de 2004: pouco mais de um ano após a divulgação do GFS, Sanjay Ghemawat, coautor do artigo sobre o GFS, junto com Jeffrey Dean, que também é funcionário da Google, publicaram o artigo *Simplified Data Processing on Large Clusters* (Dean & Ghemawat, 2004), agora sobre o modelo de programação MapReduce, que foi criado pela Google em 2003. Nesse artigo, os autores apresentaram os principais conceitos e características da tecnologia, porém, sem mencionar muitos detalhes sobre como implementá-la.
- Dezembro de 2005: o consultor de software Douglas Cutting divulgou a implementação de uma versão do MapReduce e do sistema de arquivos distribuídos com base nos artigos do GFS e do MapReduce publicados pelos funcionários da Google. Esta implementação estava integrada ao subprojeto Nutch (White, 2010), liderado por Cutting, que é o esforço da comunidade de software livre para criar um motor de busca na Web, mais especificamente um rastreador de páginas Web (*web crawler*) e um analisador de formato de documentos (*parser*). Por sua vez, o Nutch fazia parte de um projeto muito maior chamado Lucene, um programa de busca e uma API de indexação de documentos.
- Fevereiro de 2006: percebendo que enfrentava problemas similares ao de indexação de páginas da Google, a empresa Yahoo! decide contratar Cutting e investir para que ele desse andamento a seu projeto de código aberto. Nesse mesmo ano, o projeto recebe o nome de Hadoop - nome do elefante amarelo de pelúcia do filho de Cutting - deixa de ser parte integrante do Nutch e passa a ser um projeto independente da Apache Software Foundation.
- Abril de 2007: passado mais de um ano desde a contratação de Cutting, o Yahoo! anuncia ter executado com sucesso uma aplicação Hadoop em um aglomerado de 1.000 máquinas. Também nessa data, o Yahoo! passa a ser o maior contribuinte no desenvolvimento do arcabouço. Alguns anos depois, a empresa já contava com mais de 40.000 máquinas executando o Hadoop (White, 2010).
- Janeiro de 2008: o Apache Hadoop, que já se encontrava na versão 0.15.2, tem lançamento constante de versões com correção de erros e novas funcionalidades e, nesse mês, se transforma em um dos principais projetos da Apache.
- Julho de 2008: uma aplicação Hadoop em um dos aglomerados do Yahoo! quebra o recorde mundial de velocidade de processamento na ordenação de 1 *terabyte* de dados. O aglomerado era composto de 910 máquinas e executou a ordenação em 209 segundos, superando o recorde anterior que era de 297 segundos.
- Setembro de 2009: a Cloudera contrata Cutting como líder do projeto. Cloudera é uma empresa que redistribui uma versão comercial derivada do Apache

Hadoop, que integra, em um único pacote, os subprojetos mais populares do Hadoop.

- Dezembro de 2011: passado seis anos desde seu lançamento, o Apache Hadoop disponibiliza sua versão 1.0.0. Os desenvolvedores da ferramenta afirmam que essa versão possui maior confiabilidade e estabilidade em escalonamentos. Dentre os destaques dessa nova versão, cabe mencionar o uso do protocolo de autenticação de rede Kerberos, para maior segurança de rede; a incorporação do subprojeto HBase, oferecendo suporte a *BigTable*; e o suporte à interface webhdfs, que permite o acesso HTTP para leitura e escrita de dados.
- Maio de 2012: no momento em que este capítulo é escrito, a Apache faz o lançamento da primeira versão da série 2.0 do Hadoop.

Na Figura 3.1 é apresentado um cronograma no formato linha do tempo do lançamento de todas as versões disponibilizadas do Hadoop. Do lado esquerdo, estão as versões estáveis e do lado direito as versões beta. É possível perceber o grande apoio da comunidade observando a quantidade de *releases* liberadas e o curto intervalo entre elas.

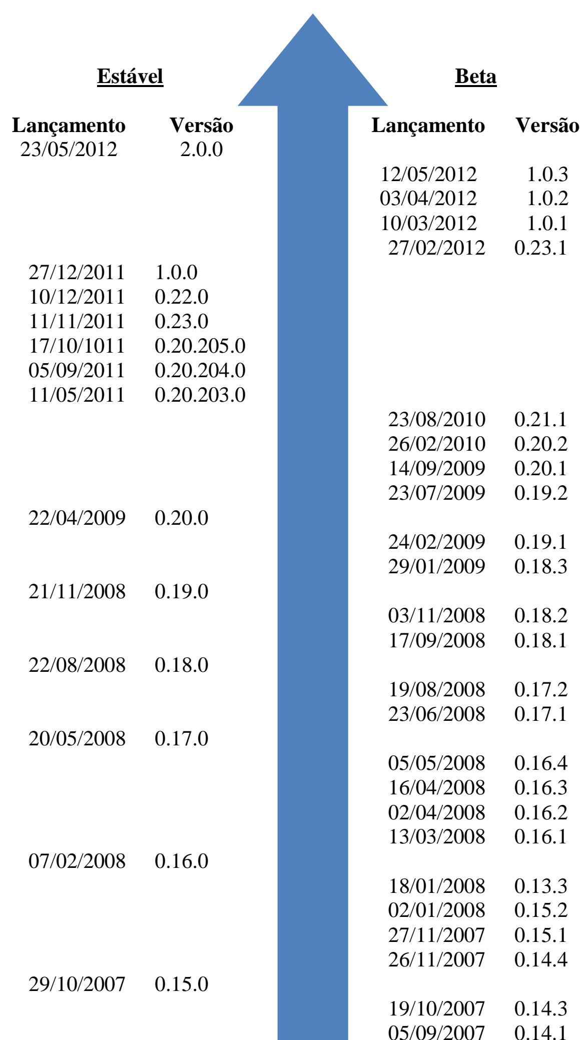

Figura 3.1. Lançamentos das versões do Apache Hadoop

3.1.2. Quem utiliza?

A eficácia obtida pelo Hadoop pode ser constatada ao verificar a quantidade de importantes empresas, de diferentes ramos, que estão usando Hadoop para fins educacionais ou de produção. Como já foi mencionado, um dos maiores desenvolvedores e contribuintes do projeto tem sido a empresa Yahoo!, entretanto, atualmente, tem sido utilizado também por outras grandes corporações. A seguir apresentaremos uma lista com algumas das principais instituições que utilizam Hadoop, todavia, uma lista mais abrangente pode ser consultada no sítio do próprio projeto[2].

Adobe (www.adobe.com): ferramentas e serviços para conteúdo digital.
Onde usa: no armazenamento e processamento de dados internos e de redes sociais. Parque: ~ 80 nós de processamento.

e-Bay (www.ebay.com): comércio eletrônico com foco em uma plataforma global de negociação (shopping popular).
Onde usa: na otimização de buscas. Parque: ~ 532 nós de processamento.

Facebook (www.facebook.com): sítio que provê serviço de rede social. Atualmente conta com mais de 845 milhões de usuários ativos.
Onde usa: análise de *log*. Parque: ~ 1.400 nós de processamento.

Last.FM (www.last.fm): rádio online agregando uma comunidade virtual com foco em música.
Onde usa: análise de *log*, análise de perfil de usuário, teste A/B, outros. Parque: ~ 64 nós de processamento.

LinkedIn (www.linkedin.com): é uma rede social de caráter profissional para compartilhar informações, ideias e oportunidades.
Onde usa: análise e busca de similaridade entre perfis de usuários. Parque: ~ 1.900 nós de processamento.

The New York Times (www.nytimes.com): uma das maiores empresa jornalística mundial.
Onde usa: conversão de imagens, armazenamento de jornais digitais. Parque: utiliza um aglomerado virtual da Amazon.

Twitter (www.twitter.com): é uma rede social e servidor para *microblogging*.
Onde usa: no armazenamento de mensagens e no processamento de informações. Parque: não divulgado.

Yahoo! (www.yahoo.com): sítio que oferece serviços de busca na Web, serviços de notícias e email.
Onde usa: no processamento de buscas, recomendações de publicidades, testes de escalabilidade. Parque: ~ 40.000 nós de processamento.

---

[2] http://wiki.apache.org/hadoop/PowerdBy

Apesar dos valores correspondentes aos parques computacionais não serem exatos, é possível vislumbrar a infraestrutura das empresas e também observar a diversidade de áreas na qual o Hadoop está sendo utilizado.

Por se tratar de um software sob a tutela da Apache Software Foundation, o Apache Hadoop segue o modelo de licenciamento da Apache[3], que é bem flexível, e permite modificações e redistribuição do código-fonte. Assim, surgiram no mercado, diversas implementações derivadas do Apache Hadoop. Cada uma dessas implementações normalmente acrescentam novas funcionalidades, aplicam especificidades de um nicho de mercado, ou ainda limitam-se a prestação de serviços como implantação, suporte e treinamento. Dentre algumas empresas com estes objetivos temos a Amazon Web Service, Cloudera, Hortonworks, KarmaSphere, Pentaho e Tresada. Atualmente, a Cloudera é chefiada por Douglas Cutting, um dos criadores do Apache Hadoop original.

3.1.3. Vantagens

O Apache Hadoop é considerado atualmente uma das melhores ferramentas para processamento de alta demanda de dados. Entre os benefícios de utilizá-lo pode-se destacar:

- Código aberto: todo projeto de software livre de sucesso tem por trás uma comunidade ativa. No projeto Apache Hadoop, essa comunidade é composta por diversas empresas e programadores independentes partilhando seus conhecimentos no desenvolvimento de melhorias, funcionalidades e documentação. Entende-se que essa colaboração remeta a uma possível melhoria na qualidade do software devido ao maior número de desenvolvedores e usuários envolvidos no processo. Um maior número de desenvolvedores é potencialmente capaz de identificar e corrigir mais falhas em menos tempo. Um número maior de usuários gera situações de uso e necessidades variadas. Também, em um trabalho cooperativo, os desenvolvedores tendem a ser mais cuidadosos em suas atividades, pois sabem que sua produção será avaliada por muitos outros profissionais. Além disso, sendo um software de código aberto, tem por princípio a garantia das quatro liberdades aos seus usuários: liberdade para executar o programa para qualquer propósito; liberdade de estudar como o programa funciona e adaptá-lo para as suas necessidades; liberdade de redistribuir cópias do programa; e liberdade para modificar o programa e distribuir essas modificações, de modo que toda a comunidade se beneficie;
- Economia: podemos apontar neste quesito 3 formas de economia. Primeiro, como já discutido no item anterior, por ser um software livre, com um esforço relativamente pequeno é possível implantar, desenvolver aplicações e executar Hadoop sem gastar com aquisição de licenças e contratação de pessoal especializado. Segundo, um dos grandes benefícios para quem faz uso do Hadoop é a possibilidade realizar o processamento da sua massa de dados utilizando máquinas e rede convencionais. Por último, existe uma outra alternativa econômica dado pela existência de serviços em nuvem, como a Amazon Elastic MapReduce (EMR), que permite a execução de aplicações

[3] http://pt.wikipedia.org/wiki/Licença_Apache

Hadoop sem a necessidade de implantar seu próprio aglomerado de máquinas, alugando um parque virtual ou simplesmente pagando pelo tempo de processamento utilizado;

- Robustez: um outro diferencial do Hadoop é que como ele foi projetado para ser executado em hardware comum, ele já considera a possibilidade de falhas frequentes nesses equipamentos e oferece estratégias de recuperação automática para essas situações. Assim, sua implementação disponibiliza mecanismos como replicação de dados, armazenamento de metadados e informações de processamento, que dão uma maior garantia para que a aplicação continue em execução mesmo na ocorrência de falhas em algum recurso;
- Escalabilidade: enquanto as demais aplicações similares apresentam dificuldade em aumentar a quantidade de máquinas utilizadas no processamento e/ou aumentar o conjunto de dados, precisando em alguns casos até reescrever todo o código-fonte da aplicação, Hadoop permite obter escalabilidade de forma relativamente simples. Mudanças no ambiente implicam em pequenas modificações em um arquivo de configuração. Dessa forma, o trabalho para preparar um ambiente contendo mil máquinas não será muito maior do que se este fosse de dez máquinas. O aumento no volume de dados só fica limitado aos recursos, espaço em disco e capacidade de processamento, disponíveis nos equipamentos do aglomerado, sem a necessidade de alteração da codificação;
- Simplicidade: Hadoop retira do desenvolvedor a responsabilidade de gerenciar questões relativas à computação paralela, tais como tolerância a falhas, escalonamento e balanceamento de carga, ficando estas a cargo do próprio arcabouço. O Hadoop descreve suas operações apenas por meio das funções de mapeamento (*Map*) e de junção (*Reduce*). Dessa forma, o foco pode ser mantido somente na abstração do problema, para que este possa ser processado no modelo de programação MapReduce.

3.1.4. Desvantagens

Como o Apache Hadoop é um arcabouço recente e em constante evolução, algumas de suas funcionalidades ainda não estão maduras o suficiente para dar suporte a todas as situações pelas quais está sendo empregado. Além disso, como desvantagem, somam-se outras dificuldades inerentes ao modelo de programação paralelo. Ressalta-se que algumas dessas desvantagens listadas a seguir estão, na medida do possível, constantemente sendo amenizadas a cada nova versão disponibilizada do arcabouço.

- Único nó mestre: a arquitetura do Hadoop tradicionalmente utiliza várias máquinas para o processamento e armazenamento dos dados, porém, contém apenas um nó mestre. Esse modelo pode se tornar restritivo a medida que essa centralidade causar empecilhos na escalabilidade e ainda criar um ponto crítico, suscetível a falha, uma vez que o nó mestre é vital para o funcionamento da aplicação.
- Dificuldade no gerenciamento do aglomerado: um dos maiores problemas sofridos pelos desenvolvedores de computação paralela ocorre no momento de depurar a execução da aplicação e de analisar os *logs* que estão distribuídos. Infelizmente, com o Hadoop, também ocorrem essas mesmas dificuldades, entretanto, existem subprojetos do ecossistema Hadoop que procuram minimizar

esse problema, como por exemplo, o arcabouço ZooKeeper, que é descrito na Seção 3.2.1.2.

Mesmo sendo uma excelente ferramenta para o processamento de dados em larga escala, existem situações para as quais o Hadoop não é a solução mais adequada, devido a alguma particularidade, como as citadas a seguir:

- Problemas não paralelizáveis ou com grande dependência entre os dados: esta é uma das maiores dificuldades para toda e qualquer aplicação que requer alto desempenho. Como o princípio básico dos programas paralelos e/ou distribuídos é dividir as tarefas em subtarefas menores, para serem processadas simultaneamente por vários recursos computacionais, a impossibilidade ou a dificuldade na extração da porção paralelizável da aplicação torna-se um grande fator de insucesso. Assim, problemas que não podem ser divididos em problemas menores e problemas com alta dependência entre os seus dados não são bons candidatos para o Hadoop. Como exemplo, podemos imaginar uma tarefa T sendo dividida em cinco subtarefas t_1, t_2, t_3, t_4 e t_5. Suponha cada tarefa t_{i+1} dependente do resultado da tarefa t_i. Dessa forma, a dependência entre as subtarefas limita ou elimina a possibilidade de sua execução em paralelo. O tempo total de execução de uma aplicação nesse contexto será equivalente (ou maior) do que se fosse executada sequencialmente;
- Processamento de arquivos pequenos: trabalho pequeno definitivamente não é tarefa para ser executada com base em um arcabouço com foco em larga escala. Os custos adicionais (*overhead*) impostos pela divisão e junção das tarefas, rotinas e estruturas gerenciais do ambiente e comunicação entre os nós de processamento, certamente irão inviabilizar sua execução;
- Problemas com muito processamento em poucos dados: problemas com regras de negócio complexas demais ou com um fluxo de execução extenso em um conjunto de dados não muito grande, também não serão bons candidatos a se tornar aplicações Hadoop. Como o foco do modelo de programação é oferecer simplicidade com as operações *Map* e *Reduce*, pode ser uma tarefa árdua tentar abstrair toda a complexidade do problema apenas nessas duas funções.

3.2. Arcabouço Apache Hadoop

Hadoop é um arcabouço de código aberto, implementado em Java e utilizado para o processamento e armazenamento em larga escala, para alta demanda de dados, utilizando máquinas comuns. Os elementos chave do Hadoop são o modelo de programação MapReduce e o sistema de arquivos distribuído HDFS. Entretanto, em meio a sua evolução, novos subprojetos, cada um para uma proposta específica da aplicação, foram incorporados ao seu ecossistema, tornando a infraestrutura do arcabouço cada vez mais completa.

3.2.1. Subprojetos do Hadoop

A grande quantidade de dados gerada por diversos segmentos trouxe a necessidade de se criar alternativas capazes de promover um processamento mais rápido e eficaz que os fornecidos pelos bancos de dados relacionais. Essa necessidade, aliada à carência de ferramentas específicas, fez com que grandes empresas experimentassem o arcabouço. Por se tratar de um projeto de código aberto, a Apache recebeu fortes estímulos dessas

empresas, que inclusive tornaram-se contribuintes no seu desenvolvimento. Ao passo que novas necessidades vinham à tona, novas ferramentas foram criadas para serem executadas sobre o Hadoop. Conforme essas ferramentas se consolidavam, passavam a integrar ou virar projetos da Apache. Foram essas muitas contribuições que fizeram com que o Hadoop tivesse uma rápida evolução, tornando-se a cada nova ferramenta incorporada, um arcabouço cada vez mais completo.

Conforme apresentado na Figura 3.2, pode-se perceber a ampla gama de subprojetos vinculados atualmente ao Hadoop e cada um com uma finalidade específica. Na camada de armazenamento de dados, temos os sistemas de arquivos distribuídos: Hadoop Distributed File System (HDFS), um dos subprojetos principais do arcabouço; e o HBase, outro sistema de arquivos, que foi desenvolvido posteriormente ao HDFS. Já na camada de processamento de dados temos disponível o modelo de programação MapReduce, que também figura como um dos principais subprojetos do Hadoop. Na camada de acesso aos dados são disponibilizadas três diferentes ferramentas: Pig, Hive e Avro. Estas ferramentas tendem a facilitar o acesso à análise e consulta dos dados, inclusive com linguagem de consulta semelhante às utilizadas em banco de dados relacionais. Por se tratar de uma aplicação paralela, dificuldades são encontradas no manuseio das aplicações, porém, nesta camada de gerenciamento, existe duas ferramentas diferentes: ZooKeeper e Chukwa.

Uma aplicação Hadoop exige no mínimo a utilização das ferramentas da camada de armazenamento e processamento, as demais podem ser adicionadas conforme haja necessidade. Além disso, para acessar esses recursos, a camada de conexão também precisa ser implementada. É por meio dessa camada que aplicações que necessitam de processamento de alto desempenho, como *Data Warehouse*, *Business Intelligence* e outras aplicações analíticas, poderão desfrutar dos benefícios do Hadoop. Cada um dos subprojetos apontados aqui são brevemente explicados nas Seções 3.2.1.1 e 3.2.1.2, que separam, respectivamente, os subprojetos centrais dos demais. Ressalta-se ainda que essa lista não contempla todos os subprojetos relacionados ao Hadoop.

3.2.1.1. Principais subprojetos

- **Hadoop Common:** contém um conjunto de utilitários e a estrutura base que dá suporte aos demais subprojetos do Hadoop. Utilizado em toda a aplicação, possui diversas bibliotecas como, por exemplo, as utilizadas para seriação de dados e manipulação de arquivos. É neste subprojeto também que são disponibilizadas as interfaces para outros sistemas de arquivos, tais como Amazon S3 e Cloudsource.
- **Hadoop MapReduce:** um modelo de programação e um arcabouço especializado no processamento de conjuntos de dados distribuídos em um aglomerado computacional. Abstrai toda a computação paralela em apenas duas funções: *Map* e *Reduce*.
- **Hadoop Distributed File System (HDFS):** um sistema de arquivos distribuído nativo do Hadoop. Permite o armazenamento e transmissão de grandes conjuntos de dados em máquinas de baixo custo. Possui mecanismos que o caracteriza como um sistema altamente tolerante a falhas.

3.2.1.2. Outros subprojetos

- **Avro[4]:** sistema de seriação de dados baseado em *schemas*. Sua composição é dada por um repositório de dados persistentes, um formato compacto de dados binários e suporte a chamadas remotas de procedimentos (RPC).
- **Chukwa[5]:** sistema especialista em coleta e análise de *logs* em sistemas de larga escala. Utiliza HDFS para armazenar os arquivos e o MapReduce para geração de relatórios. Possui como vantagem um *kit* auxiliar de ferramentas, muito poderoso e flexível, que promete melhorar a visualização, monitoramento e análise dos dados coletados.
- **Hbase[6]:** banco de dados criado pela empresa Power Set em 2007, tendo posteriormente se tornando um projeto da Apache Software Foundation. Considerado uma versão de código aberto do banco de dados *BigTable,* criada pela Google, é um banco de dados distribuído e escalável que dá suporte ao armazenamento estruturado e otimizado para grandes tabelas.
- **Hive[7]:** arcabouço desenvolvido pela equipe de funcionários do Facebook, tendo se tornado um projeto de código aberto em agosto de 2008. Sua principal funcionalidade é fornecer uma infraestrutura que permita utilizar *Hive QL*, uma linguagem de consulta similar a SQL bem como demais conceitos de dados relacionais tais como tabelas, colunas e linhas, para facilitar as análises complexas feitas nos dados não relacionais de uma aplicação Hadoop.
- **Pig[8]:** uma linguagem de alto nível orientada a fluxo de dados e um arcabouço de execução para computação paralela. Sua utilização não altera a configuração do aglomerado Hadoop, pois é utilizado no modo *client-side*, fornecendo uma linguagem chamada *Pig Latin* e um compilador capaz de transformar os programas do tipo Pig em sequências do modelo de programação MapReduce.
- **ZooKeeper[9]:** arcabouço criado pelo Yahoo! em 2007 com o objetivo de fornecer um serviço de coordenação para aplicações distribuídas de alto desempenho, que provê meios para facilitar as seguintes tarefas: configuração de nós, sincronização de processos distribuídos e grupos de serviço.

---

[4] Avro: http://avro.apache.org
[5] Chukwa: http://incubator.apache.org/chukwa
[6] Hbase: http://hbase.apache.org
[7] Hive: http://hive.apache.org
[8] Pig: http://pig.apache.org
[9] ZooKeeper: http://zookeeper.apache.org

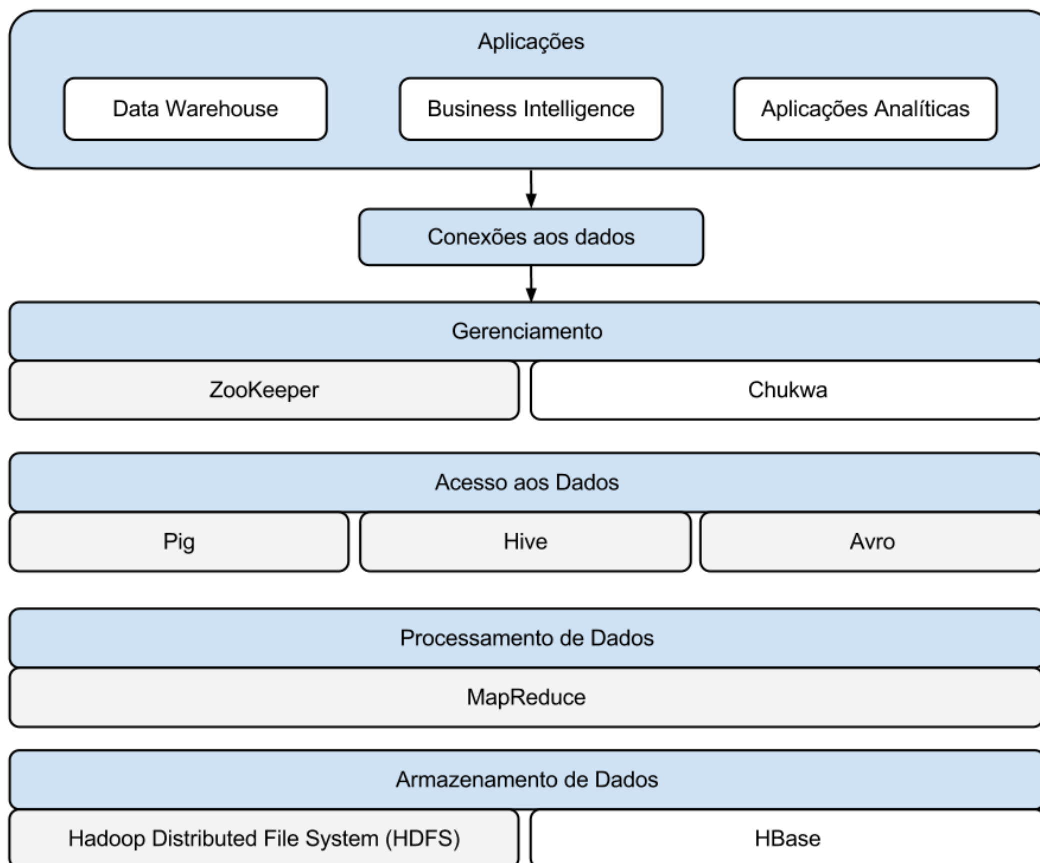

Figura 3.2. Subprojetos do Apache Hadoop
(Adaptada de: Rogers, 2011)

3.2.2. Componentes do Hadoop

Uma execução típica de uma aplicação Hadoop em um aglomerado utiliza cinco processos diferentes: NameNode, DataNode, SecondaryNameNode, JobTracker e TaskTracker. Os três primeiros são integrantes do modelo de programação MapReduce, e os dois últimos do sistema de arquivo HDFS. Os componentes NameNode, JobTracker e SecondaryNameNode são únicos para toda a aplicação, enquanto que o DataNode e JobTracker são instanciados para cada máquina.

- NameNode: tem como responsabilidade gerenciar os arquivos armazenados no HDFS. Suas funções incluem mapear a localização, realizar a divisão dos arquivos em blocos, encaminhar os blocos aos nós escravos, obter os metadados dos arquivos e controlar a localização de suas réplicas. Como o NameNode é constantemente acessado, por questões de desempenho, ele mantém todas as suas informações em memória. Ele integra o sistema HDFS e fica localizado no nó mestre da aplicação, juntamente com o JobTracker.

- DataNode: enquanto o NameNode gerencia os blocos de arquivos, são os DataNodes que efetivamente realizam o armazenamento dos dados. Como o HDFS é um sistema de arquivos distribuído, é comum a existência de diversas instâncias do DataNode em uma aplicação Hadoop, para que eles possam distribuir os blocos de arquivos em diversas máquinas. Um DataNode poderá

armazenar múltiplos blocos, inclusive de diferentes arquivos. Além de armazenar, eles precisam se reportar constantemente ao NameNode, informando quais blocos estão guardando bem como todas as alterações realizadas localmente nesses blocos.

- JobTracker: assim como o NameNode, o JobTracker também possui uma função de gerenciamento, porém, nesse caso, o controle é realizado sobre o plano de execução das tarefas a serem processadas pelo MapReduce. Sua função então é designar diferentes nós para processar as tarefas de uma aplicação e monitorá-las enquanto estiverem em execução. Um dos objetivos do monitoramento é, em caso de falha, identificar e reiniciar uma tarefa no mesmo nó ou, em caso de necessidade, em um nó diferente.
- TaskTracker: processo responsável pela execução de tarefas MapReduce. Assim como os DataNodes, uma aplicação Hadoop é composta por diversas instâncias de TaskTrackers, cada uma em um nó escravo. Um TaskTracker executa uma tarefa *Map* ou uma tarefa *Reduce* designada a ele. Como os TaskTrackers rodam sobre máquinas virtuais, é possível criar várias máquinas virtuais em uma mesma máquina física, de forma a explorar melhor os recursos computacionais.
- SecondaryNameNode: utilizado para auxiliar o NameNode a manter seu serviço, e ser uma alternativa de recuperação no caso de uma falha do NameNode. Sua única função é realizar pontos de checagem (*checkpointing*) do NameNode em intervalos pré-definidos, de modo a garantir a sua recuperação e atenuar o seu tempo de reinicialização.

Na Figura 3.3 pode-se observar de forma mais clara como os processos da arquitetura do Hadoop estão interligados. Inicialmente nota-se uma separação dos processos entre os nós mestre e escravos. O primeiro contém o NameNode, o JobTracker e possivelmente o SecondaryNameNode. Já o segundo, comporta em cada uma de suas instâncias um TaskTracker e um DataNode, vinculados respectivamente ao JobTracker e ao NameNode do nó mestre.

Um cliente de uma aplicação se conecta ao nó mestre e solicita a sua execução. Nesse momento, o JobTracker cria um plano de execução e determina quais, quando e quantas vezes os nós escravos processarão os dados da aplicação. Enquanto isso, o NameNode, baseado em parâmetros já definidos, fica encarregado de armazenar e gerenciar as informações dos arquivos que estão sendo processados. Do lado escravo, o TaskTracker executa as tarefas a ele atribuídas, que ora podem ser *Map* ora *Reduce*, e o DataNode armazena um ou mais blocos de arquivos. Durante a execução, o nó escravo também precisa se comunicar com o nó mestre, enviando informações de sua situação local.

Paralelamente a toda essa execução, o SecondaryNameNode registra pontos de checagem dos arquivos de *log* do NameNode, para a necessidade de uma possível substituição no caso do NameNode falhar. Outros detalhes da funcionalidade desses processos são apresentados nas Seções 3.3 e 3.4.

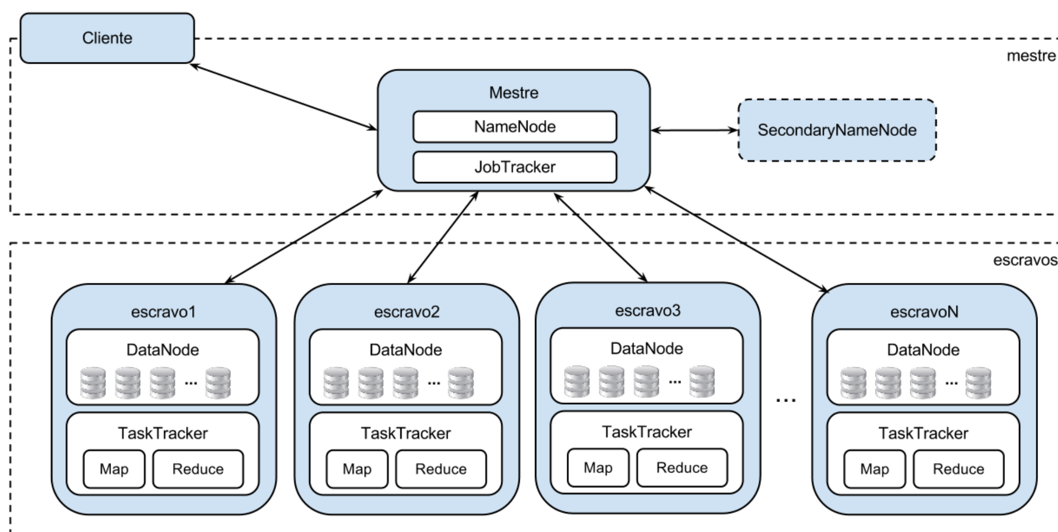

Figura 3.3. Processos do Hadoop

3.2.3. Formas de execução

Embora uma aplicação Hadoop seja tipicamente executada em um conjunto de máquinas, ela pode também ser executada em um único nó. Essa possibilidade permite adotar configurações simplificadas para as fases iniciais de implementação e testes, visto que depurar aplicações distribuídas não é algo trivial. Posteriormente, outras configurações mais sofisticadas podem ser utilizadas para usufruir de todas as vantagens oferecidas pelo arcabouço. Chuck Lam (Lam, 2010) cita três modos possíveis de execução: modo local (*standalone mode*), modo pseudo-distribuído (*pseudo-distributed mode*) e modo completamente distribuído (*fully distributed mode*). Para alternar entre essas configurações é necessária a edição de três arquivos: *core-site.xml*, *hdfs-site.xml* e *mapred-site.xml*.

Modo Local

Hadoop é por padrão configurado para ser executado no modo local. Dessa maneira, se essa for a sua opção escolhida, os parâmetros nos arquivos de configuração não precisam ser alterados. Esse modo é o mais recomendado para a fase de desenvolvimento, onde normalmente ocorre a maior incidência de erros, sendo necessária a realização de vários testes da execução. Nessa configuração, todo o processamento da aplicação é executado apenas na máquina local. Por isso, não é necessário que os arquivos sejam carregados no HDFS, visto que, os dados não são distribuídos. Dessa forma simplificada, fica mais fácil para o usuário realizar a depuração de seu código, aumentando sua produtividade.

Modo pseudo-distribuído

Uma segunda alternativa para executar uma aplicação Hadoop é o modo pseudo-distribuído. Nesse modo são aplicadas todas as configurações, semelhantes às necessárias para execução em um aglomerado, entretanto, toda a aplicação é processada em modo local, por isso o termo pseudo-distribuído ou também chamado "*cluster*" de uma máquina só. Embora não seja executado realmente em paralelo, esse modo permite a sua simulação, pois utiliza todos os processos de uma execução paralela efetiva: NameNode, DataNode, JobTracker, TaskTracker e SecondaryNameNode.

Quadro 3.1. Configuração do arquivo core-site.xml no modo pseudo-distribuído

```
1. <!-- core-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.    <property>
5.       <name>fs.default.name</name>
6.       <value>hdfs://localhost:9000</value>
7.       <description>The name of the default file system. A URI
           whose scheme and authority determine the FileSystem
           implementation.</description>
8.    </property>
9. </configuration>
```

No Quadro 3.1 é mostrada a configuração do arquivo core-site.xml, utilizado para especificar a localização do NameNode. As informações mais importantes desse arquivo estão delimitadas pelas *tags* XML <name> e <value>, que estão respectivamente nas linhas 5 e 6. A primeira define o nome da variável a ser editada, *fs.default.name*, e a segunda, o valor atribuído a ela. Ainda na linha 6, observamos que o valor da variável foi composto pelo protocolo hdfs, pelo *hostname* e pela porta de localização definida para o HDFS. As demais linhas trazem comentários e *tags* necessárias para a estruturação do arquivo XML.

Quadro 3.2. Configuração do arquivo mapred-site.xml no modo pseudo-distribuído

```
1. <!-- mapred-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.   <property>
5.      <name>mapred.job.tracker</name>
6.      <value>localhost:9001</value>
7.      <description>The host and port that the MapReduce job
          tracker runs at.</description>
8.   </property>
9. </configuration>
```

No Quadro 3.2 podemos visualizar o conteúdo do arquivo mapred-site.xml. Nesse exemplo, na linha 5, fica definido na *tag* <name> o nome da variável *mapred.job.tracker*. Seu valor é explicitado na linha 6, e é composto pelo *hostname* e pela porta onde o JobTracker será executado. Assim como no quadro anterior, as demais *tags* apresentam uma conotação auxiliar e/ou estrutural para o arquivo.

Quadro 3.3 - Configuração do arquivo hdfs-site.xml no modo pseudo-distribuído

```
1. <!-- hdfs-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.    <property>
5.       <name>dfs.replication</name>
6.       <value>1</value>
7.       <description>The actual number of replications can be
           specified when the file is created.</description>
8.    </property>
9. </configuration>
```

A terceira configuração apresentada no Quadro 3.3 representa o conteúdo do arquivo hdfs-site.xml, que é utilizado para definir o número de réplicas de cada bloco de arquivo armazenado no HDFS. A *tag* <name> dessa especificação, observada na linha 5, define o nome da variável como *dfs.replication*. Já a *tag* <value>, na linha 6, indica o valor correspondente ao número de réplicas. Esse último parâmetro possui por padrão o valor 3, porém, nesse exemplo, foi alterado para 1, dado que estamos no modo pseudo-distribuído que é executado em apenas um nó.

Além das configurações já apresentadas, precisamos ainda indicar a localização do SecondaryNameNode e dos nós escravos. Essa localização é dada pelo endereço de rede ou pelo apelido desses recursos nos respectivos arquivos *masters* e *slaves*. Sabemos que no modo pseudo-distribuído estamos simulando uma execução distribuída, dessa forma, para esse modo, esses locais serão sempre os mesmos. O arquivo *masters*, está representado no Quadro 3.4 e o arquivo *slaves*, no Quadro 3.5.

Quadro 3.4 - Conteúdo do arquivo *masters*

```
localhost
```

Quadro 3.5 - Conteúdo do arquivo *slaves*

```
localhost
```

Modo completamente distribuído

Por fim, o terceiro e último modo de execução é utilizado para o processamento distribuído da aplicação Hadoop em um aglomerado de computadores real. Nessa opção, como no modo pseudo-distribuído, também é necessário editar os três arquivos de configuração, definindo parâmetros específicos e a localização do SecondaryNameNode e dos nós escravos. Todavia, como nesse modo temos diversos computadores, devemos indicar quais máquinas irão efetivamente executar cada componente.

Nos Quadros 3.6 e 3.7 apresentamos exemplos de configuração do NameNode e do JobTracker, respectivamente. No Quadro 3.8, na linha 6, definimos o fator de replicação para os blocos nos DataNodes. Apresentamos mais detalhes sobre esse fator de replicação na Seção 3.3, específica do HDFS.

Quadro 3.6 - Configuração do arquivo core-site.xml no modo completamente distribuído

```
1. <!-- core-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.  <property>
5.     <name>fs.default.name</name>
6.     <value>hdfs://master:9000</value>
7.     <description> The name of the default file system. A URI
         whose scheme and authority determine the FileSystem
         implementation. </description>
8.  </property>
9.</configuration>
```

Quadro 3.7 - Configuração do arquivo mapred-site-site.xml no modo completamente distribuído

```
1. <!-- mapred-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.   <property>
5.       <name>mapred.job.tracker</name>
6.       <value>master:9001</value>
7.       <description> The host and port that the MapReduce job
           tracker runs at.</description>
8.  </property>
9. </configuration>
```

Quadro 3.8 - Configuração do arquivo hdfs-site.xml no modo completamente distribuído

```
1. <!-- hdfs-site.xml -->
2. <?xml version="1.0"?>
3. <configuration>
4.   <property>
5.      <name>dfs.replication</name>
6.      <value>3</value>
7.      <description> The actual number of replications can be
          specified when the file is created.</description>
8.   </property>
9. </configuration>
```

No arquivo *masters*, apresentado no Quadro 3.9, indicamos *maquina_espelho* que é o apelido (*alias*) para o endereço de rede (número IP) da localização do SecondaryNameNode. Já o arquivo *slaves*, Quadro 3.10, contém em cada uma de suas linhas, o apelido de cada uma das máquinas que servirão de escravas para o aglomerado Hadoop. São essas máquinas que posteriormente armazenarão os arquivos de dados e processarão a aplicação. Para usar esses apelidos é necessário escrever no arquivo */etc/hosts* do sistema operacional, em cada linha, como no Quadro 3.11, uma combinação “apelido” “endereço de rede” para cada uma das máquinas.

Quadro 3.9 - Conteúdo do arquivo *masters*

```
maquina_espelho
```

Quadro 3.10 - Conteúdo do arquivo *slaves*

```
maquina_escrava1
maquina_escrava2
maquina_escrava3
maquina_escrava4
maquina_escrava5
```

A simplicidade de preparação para um ambiente de execução do Hadoop pode ser observada pelas pequenas modificações realizadas em seus arquivos de configuração. Com um mínimo de esforço pode-se modificar totalmente seu modo de execução, incluir ou retirar máquinas escravas do aglomerado ou trocar a localização de processos de segurança.

Quadro 3.11 - Conteúdo do arquivo */etc/hosts*

```
maquina_espelho   192.168.108.1
maquina_escrava1  192.168.108.101
maquina_escrava2  192.168.108.102
maquina_escrava3  192.168.108.103
maquina_escrava4  192.168.108.104
maquina_escrava5  192.168.108.105
```

3.3. HDFS

Sistema de arquivos distribuídos

A manipulação dos arquivos em um computador é realizada pelo sistema operacional instalado na máquina por meio de um sistema gerenciador de arquivos. Esse sistema possui um conjunto de funcionalidades, tais como: armazenamento, organização, nomeação, recuperação, compartilhamento, proteção e permissão de acesso aos arquivos. Todo o gerenciamento deve ocorrer da forma mais transparente possível aos usuários, ou seja, o sistema de arquivos deve omitir toda a complexidade arquitetural da sua estrutura não exigindo do seu usuário muito conhecimento para operá-lo.

Um sistema de arquivos distribuído possui as mesmas características que um sistema de arquivos convencional, entretanto, deve permitir o armazenamento e o compartilhamento desses arquivos em diversos hardwares diferentes, que normalmente estão interconectados por meio de uma rede. O sistema de arquivos distribuído também deve prover os mesmos requisitos de transparência a seus usuários, inclusive permitindo uma manipulação remota dos arquivos como se eles estivessem localmente em suas máquinas. Além disso, deve oferecer um desempenho similar ao de um sistema tradicional e ainda prover escalabilidade.

Assim, existem algumas estruturas de controle exclusivas, ou mais complexas, que devem ser implementadas em um sistema de arquivos distribuído. Entre essas, podemos citar algumas imprescindíveis para o seu bom funcionamento:

- Segurança: no armazenamento e no tráfego das informações, garantindo que o arquivo não seja danificado no momento de sua transferência, no acesso às informações, criando mecanismos de controle de privacidade e gerenciamento de permissões de acesso;
- Tolerância a falhas: deve possuir mecanismos que não interrompam o sistema em casos de falhas em algum nó escravo;
- Integridade: o sistema deverá controlar as modificações realizadas no arquivo, como por exemplo, alterações de escrita e remoção, permitindo que esse seja modificado somente se o usuário tiver permissão para tal;
- Consistência: todos os usuários devem ter a mesma visão do arquivo;
- Desempenho: embora possua mais controles a serem tratados que o sistema de arquivos convencional, o desempenho do sistema de arquivos distribuído deve ser alto, uma vez que provavelmente deverá ser acessado por uma maior gama de usuários.

Atualmente há diversas implementações de sistemas de arquivos distribuídos, algumas comerciais e outras de software livre, tais como: GNU Cluster File System

(GlusterFS)[10] da empresa Red Hat; Moose File System (MooseFS)[11], desenvolvido pela Gemius SA; Lustre[12], originário da Carnegie Mellon University, atualmente é mantido pela Sun Microsystems; CODA[13], também desenvolvido na Carnegie Mellon University; General Parallel File System (GPFS)[14] e OpenAFS[15] da IBM, esse último derivado do Andrew File System (AFS) que também foi desenvolvido na Carnegie Mellon University; e dois outros mais conhecidos pela comunidade, que faremos uma breve descrição a seguir, Network File System (NFS) e Google File System (GFS).

O NFS é um sistema de arquivos distribuído, desenvolvido pela Sun Microsystems no início dos anos 80. O objetivo desse sistema é garantir uma utilização transparente do usuário aos arquivos, sem que ele possa distinguir se o acesso está sendo feito de forma local ou remotamente, formando assim uma espécie de diretório virtual. O NFS possui uma estrutura cliente/servidor, onde os clientes utilizam os arquivos por meio de acesso remoto, enquanto o servidor tem a responsabilidade de manter esses arquivos disponíveis. A comunicação entre cliente e servidor no NFS ocorre por meio do mecanismo RPC, que por sua vez, é executado sobre a pilha de protocolos TCP/IP e UDP/IP. Embora seja um dos sistemas de arquivos distribuídos mais antigos, o NFS continua sendo muito utilizado, principalmente em sistemas operacionais derivados do Unix.

Já o GFS, desenvolvido pela Google em 2003, conforme mencionado na Seção 3.1.1, foi desenvolvido na linguagem C++ e tem como principal característica o suporte à distribuição de quantidades massivas de dados. A arquitetura desse sistema segue o padrão mestre/escravo; antes de armazenar os dados, ele faz uma divisão dos arquivos em blocos menores, que são disseminados aos nós escravos. Atualmente, aplicações como YouTube, Google Maps, Gmail, Blogger, entre outras, geram *petabytes* de dados que são armazenados nesse sistema. Foi do GFS a inspiração para o sistema de arquivos distribuído do Hadoop, o HDFS.

HDFS

O Hadoop Distributed File System (HDFS) é um sistema de arquivos distribuído integrado ao arcabouço Hadoop. Esse sistema teve forte inspiração no GFS da Google, entretanto, uma das diferenças deve-se ao fato de que o HDFS é um arcabouço de código aberto, e foi implementado na linguagem Java. O HDFS também oferece suporte ao armazenamento e ao processamento de grandes volumes de dados em um agrupamento de computadores heterogêneos de baixo custo. Essa quantidade de dados pode chegar à ordem de *petabytes*, quantidade que não seria possível armazenar em um sistema de arquivos tradicional.

O número de máquinas utilizadas em um HDFS é uma grandeza diretamente proporcional à probabilidade de uma dessas máquinas vir a falhar, ou seja, quanto mais

[10] GlusterFS: http://www.gluster.org

[11] MooseFS: http://www.moosefs.org

[12] LUSTRE: http://www.lustre.org

[13] CODA: http://www.coda.cs.cmu.edu

[14] GPFS: http://www-03.ibm.com/systems/software/gpfs

[15] OpenFS: http://www.openafs.org

máquinas, maior a chance de acontecer algum erro em uma delas. Para contornar essa situação, o HDFS tem como princípio promover tolerância, detecção e recuperação automática de falhas. Essas funcionalidades permitem que, no caso de alguma máquina do aglomerado vir a falhar, a aplicação como um todo não seja interrompida, pois o processamento que estava sendo realizado nessa máquina poderá ser reiniciado, ou no pior caso, repassado para uma outra máquina disponível. Tudo isso de forma transparente ao usuário.

Comandos HDFS

Para iniciar os trabalhos em um aglomerado Hadoop é necessário formatar o HDFS no intuito de prepará-lo para receber os dados de sua aplicação. Essa ação pode ser realizada por meio do comando *hadoop namenode -format,* executado na máquina onde se encontra o NameNode. Um exemplo dessa ação pode ser observado a seguir:

```
bin/hadoop namenode -format
```

Embora possa ser manipulada por diversas interfaces, uma das formas comumente utilizada para manipular o HDFS é por linha de comando. Nesta interface é possível realizar várias operações, como leitura, escrita, exclusão, listagem, criação de diretório, etc. com comandos similares ao do Linux, porém iniciados pelo prefixo "hadoop fs". A sintaxe dos comandos segue a seguinte estrutura:

```
hadoop fs -comando [argumentos]
```

A listagem, a explicação e os argumentos válidos para todos os comandos do HDFS podem ser consultados executando o seguinte comando:

```
hadoop fs -help
```

Para entender o funcionamento de um comando específico, pode passá-lo como argumento. No exemplo do Quadro 3.12 é apresentada a consulta para o comando *dus*, e o resultado retornado:

Quadro 3.12. Saída produzida pela linha de comando hadoop fs -help dus

```
[hadoop_user@localhost ~]# hadoop fs –help dus
-dus <path>:  Show the amount of space, in bytes, used by the files
that match the specified file pattern. Equivalent to the unix command
"du -sb" The output is in the form name(full path) size(in bytes)
```

Antes de iniciar uma aplicação Hadoop no modo pseudo-distribuído ou completamente distribuído é necessário que os dados que serão utilizados já estejam armazenados no HDFS. Desta forma, o usuário precisa copiar os arquivos de dados da sua máquina local para o HDFS. No exemplo a seguir está explicitado o comando para carregar no HDFS o arquivo *meuarquivo.txt*.

```
hadoop fs -put meuarquivo.txt /user/hadoop_user
```

Nesse exemplo foi utilizado o comando *-put*, informado como parâmetros o nome do arquivo, bem como o diretório *user/hadoop_user*, para o qual ele será adicionado. Por padrão, o HDFS possui um diretório com o nome do usuário dentro do diretório */user*. Nesse exemplo o usuário é o *hadoop_user*. Se acaso o usuário desejar criar outros diretórios, o comando que realiza essa ação é o *mkdir*, conforme exemplo a seguir, onde será criado o diretório *arquivos_hadoop*.

```
hadoop fs –mkdir arquivos_hadoop
```

Nesse caso não foi mencionado o caminho completo do local onde o diretório deverá ser criado, assim, quando essa informação for omitida, o arquivo será armazenado no diretório padrão *user/hadoop_user*. Portanto, o caminho completo para acesso dos arquivos inseridos no diretório *arquivos_hadoop* será *user/hadoop_user/arquivos_hadoop*.

Para listar todos os arquivos e diretórios contidos no diretório raiz, que no caso é */user/hadoop_user*, executamos o seguinte comando:

```
hadoop fs –ls
```

Para listar arquivos, diretórios e os subdiretórios, deve-se acrescentar o comando de recursividade, como no exemplo a seguir:

```
hadoop fs -lsr
```

A seguir, no Quadro 3.13, é apresentado o resultado da ação dos dois comandos de listagem de arquivos.

Quadro 3.13. Saída produzida pelas linhas de comando hadoop fs -ls e hadoop fs -lsr

```
[hadoop_user@localhost ~]# hadoop fs -ls /
Found 1 items
drwxr-xr-x   - hadoop_user supergroup        0 2012-03-14 10:15 /user


[hadoop_user@localhost ~]# hadoop fs -lsr /
drwxr-xr-x  - hadoop_user supergroup   0 2012-03-14 10:15 /user
drwxr-xr-x  - hadoop_user supergroup   0 2012-03-14 11:21
/user/hadoop_user
-rw-r--r--  3 hadoop_user supergroup 264 2012-03-14 11:24
/user/hadoop_user/meuarquivo.txt
```

A saída apresenta, em ordem, as seguintes informações: permissões de acesso, fator de replicação, dono do arquivo, grupo ao qual o dono pertence, tamanho, data e hora da última modificação e caminho do arquivo e/ou diretório. O fator de replicação é aplicado somente à arquivos, e por tal motivo, nos diretórios a quantidade é marcada por um traço.

A partir do momento em que os arquivos estão armazenados no HDFS, já são passíveis de serem submetidos ao processamento de uma aplicação Hadoop. Se após a execução, for necessário copiar novamente os arquivos ao sistema local, isto poderá ser feito pelo comando *-get*, conforme o seguinte exemplo:

```
hadoop fs -get meuarquivo.txt localfile
```

Nesse exemplo, após o comando, como primeiro argumento, deve ser passado o nome do arquivo que deseja copiar do HDFS, com o seu respectivo caminho. O segundo parâmetro é o diretório local onde se deseja colocar o arquivo copiado.

Como pode ser visto, a interface de linha de comando pode ser utilizada sem muita dificuldade, principalmente para os conhecedores de Linux. Entretanto, caso essa interface não seja adequada, o usuário pode optar por outras alternativas providas pelo HDFS, podendo até mesmo usar a API Java para realizar essa manipulação. Perceba que em nenhum momento falamos de comandos específicos para um sistema de arquivos distribuídos como para tratar tolerância a falhas, balanceamento de carga e disponibilidade, pois são todas ações tratadas pelo próprio arcabouço.

Divisão em blocos

Existem arquivos que devido ao seu grande tamanho, não podem ser armazenados em um único disco rígido. Esta situação fica mais evidente quando nos referimos aos arquivos de aplicações “Big Data”. Uma alternativa nesse caso é criar uma forma de dividir esses grandes arquivos e distribuí-los em um aglomerado de máquinas. Embora funcional, essa tarefa seria muito difícil de ser implementada sem a utilização de um recurso específico, como o HDFS. Com esse arcabouço, toda a questão estrutural relativa a distribuição dos arquivos é feita de forma implícita, devendo apenas que o desenvolvedor aponte corretamente os parâmetros de configuração. Como exemplo ver os Quadros 3.9 e 3.10.

O HDFS adota a estratégia de que antes de armazenar os arquivos, estes sejam submetidos a um procedimento de divisão em uma sequência de blocos de tamanho fixo. O tamanho padrão definido no arcabouço é 64 Mb, podendo ser alterado se necessário. Esse tamanho é muito superior ao dos sistemas de arquivos tradicionais, que usa blocos de 512 *bytes*. Dessa forma, somente depois de dividido é que esses arquivos são distribuídos para os diversos nós escravos. Uma outra característica interessante do HDFS é que, no caso de um dado não ocupar todo o tamanho do bloco reservado a ele, o espaço restante não é perdido, podendo ser utilizado por outros dados.

Arquitetura

O HDFS é implementado sobre a arquitetura mestre/escravo, possuindo no lado mestre uma instância do NameNode e em cada escravo uma instância do DataNode, como visto na Figura 3.3. Em um aglomerado Hadoop podemos ter centenas ou milhares de máquinas escravas, e dessa forma, elas precisam estar dispostas em diversos armários (*racks*). Nesse texto, a palavra armário é utilizada como um conjunto de máquinas alocadas em um mesmo espaço físico e interligadas por um comutador (*switch*). Por questões estratégicas, que serão discutidas na próxima seção, o HDFS organiza a armazenagem dos blocos dos arquivos, e suas réplicas, em diferentes máquinas e armários. Assim, mesmo ocorrendo uma falha em um armário inteiro, o dado pode ser recuperado e a aplicação não precisaria ser interrompida.

O NameNode é o componente central do HDFS, assim, é recomendável ser implantado em um nó exclusivo, e preferencialmente o nó com melhor desempenho do aglomerado. Ainda por questões de desempenho, o NameNode mantém todas suas informações em memória. Para desempenhar seu papel de gerenciar todos os blocos de arquivos, o NameNode possui duas estruturas de dados importantes: o FsImage e o EditLog. O primeiro arquivo é o responsável por armazenar informações estruturais dos blocos, como o mapeamento e *namespaces* dos diretórios e arquivos, e a localização das réplicas desses arquivos. O segundo, EditLog, é um arquivo de *log* responsável por armazenar todas as alterações ocorridas nos metadados dos arquivos.

Ao iniciar uma instância do NameNode, suas tarefas iniciais são: realizar a leitura do último FsImage, e aplicar as alterações contidas no EditLog. Terminada essa operação, o estado do HDFS é atualizado, e o arquivo de *log* é esvaziado, para manter apenas as novas alterações. Esse procedimento ocorre somente quando o NameNode é iniciado, e por tal motivo, passado muito tempo de sua execução, o EditLog tende a ficar muito extenso, e pode afetar o desempenho do sistema, ou ainda, acarretar muitas

operações na próxima inicialização do NameNode. Para que isto não ocorra, existe um componente assistente ao NameNode, chamado SecondaryNameNode.

Mesmo não sendo exatamente um backup do NameNode, no caso desse vir a ser interrompido, uma solução é tornar o SecondaryNameNode o NameNode primário, como uma forma de prevenção de interrupção do sistema. O SecondaryNameNode tem como principal função realizar a junção entre o FsImage e EditLog criando pontos de checagem, de modo a limpar o arquivo de *log*. Essa operação é feita em intervalos de tempo definidos na configuração do sistema. Dessa forma, como o SecondaryNameNode não é atualizado em tempo real, esse atraso poderia ocasionar a perda de dados.

Enquanto o nó mestre é o responsável por armazenar os metadados dos arquivos, os nós escravos são os responsáveis pelo armazenamento físico dos dados. São nesses escravos que temos os DataNodes. Em uma aplicação Hadoop, cada nó escravo contém um DataNode, que trabalha em conjunto com um TaskTracker, sendo o primeiro para armazenamento e o segundo para processamento dos dados.

A primeira comunicação entre o mestre e o escravo ocorre quando o DataNode é registrado no NameNode, que pode ocorrer no momento da inicialização ou quando esse for reinicializado. Todo esse procedimento de registro é armazenado no arquivo FsImage do NameNode. Após essa interação, o DataNode precisa ainda periodicamente se comunicar com o NameNode, enviando informações estatísticas dos blocos que está armazenando, bem como informações de suas alterações locais. São nesses momentos de interação que se torna possível ao NameNode definir quais nós deverão armazenar quais blocos. Se acaso o NameNode não conseguir receber informações do DataNode, é solicitado que esse DataNode seja novamente registrado.

Replicação de dados

Além de dividir os arquivos em blocos, o HDFS ainda replica esses blocos na tentativa de aumentar a segurança. Por padrão, um bloco do HDFS possui três réplicas alocadas em diferentes nós, podendo essa quantidade ser configurada. Ainda existe uma recomendação, por questão de confiabilidade e desempenho, de alocar duas réplicas no mesmo armário, porém em nós distintos, e a outra réplica em um armário diferente. Como tipicamente a velocidade de comunicação entre máquinas de um mesmo armário é maior que em armários diferentes, por questão de desempenho, no momento de selecionar uma réplica para ser substituída em um processo, o HDFS dá preferência à réplica pertencente ao mesmo armário.

Na Figura 3.4 é demonstrado o processo de replicação de blocos. Nesse exemplo copiaremos para o HDFS dois arquivos, *arquivoA* e *arquivoB*, de tamanho e formato distintos. Antes de enviá-los aos escravos, cada um dos arquivos é subdividido em *N* blocos de tamanho fixo de 64 Mb, podendo o último ser menor por alocar o final do arquivo. Assim, para o *arquivoA* teremos os blocos *a1*, *a2*, *a3*, *a4*, ..., *aN*, e para o *arquivoB* os blocos *b1*, *b2*, ..., *bN*, conforme necessidade. Além disso, os blocos ainda serão replicados para serem distribuídos aos nós escravos. Para cada bloco foi definido um fator de replicação de 3 (três) unidades, então, observamos na mesma figura que o bloco *a1* (quadrado com linha sólida) foi armazenado no *escravoX1* e as suas réplicas (quadrado com linha tracejada) no *escravoX2* e no *escravoY3*. Para uma maior garantia de disponibilidade dos blocos, as réplicas ainda são propositalmente colocadas em

armário*s* diferentes, como pode ser confirmado observando as réplicas do bloco *a1* no *armárioX*, e no *armárioY*. Toda essa lógica também é aplicada a todos os demais blocos.

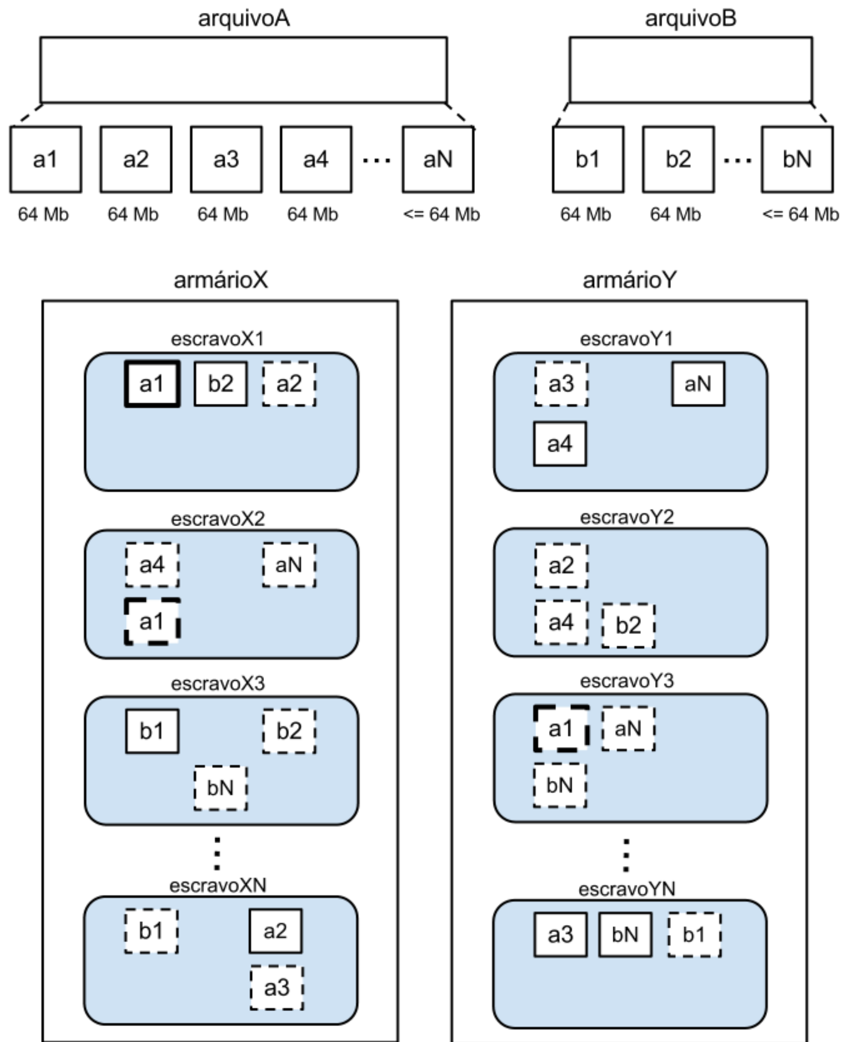

Figura 3.4. Replicação de blocos de dados

O maior benefício com a replicação é a obtenção de maior tolerância a falhas e confiabilidade dos dados, pois no caso de um nó escravo vir a falhar, o processamento passará a ser feito por outra máquina que contenha a réplica desse bloco, sem haver a necessidade de transferência de dados e a interrupção da execução da aplicação. Tudo isso é feito de forma transparente, pois o Hadoop oferece mecanismos para reiniciar o processamento sem que os demais nós percebam a falha ocorrida. No contexto de uma falha ocorrerá uma diminuição da quantidade de réplicas de um bloco. Então, para retomar a sua margem de confiabilidade, o NameNode consulta os metadados sobre os DataNodes falhos e reinicia o processo de replicação em outros DataNodes, para garantir o seu fator mínimo.

3.4. Hadoop MapReduce

O paradigma de programação MapReduce implementado pelo Hadoop se inspira em duas funções simples (*Map* e *Reduce*) presentes em diversas linguagens de programação funcionais. Uma das primeiras linguagens a implementar os conceitos dessas funções foi LISP. Essas funções podem ser facilmente explicadas de acordo com suas implementações originais, conforme os exemplos a seguir, onde serão usados pseudocódigos para ilustrar tais funções.

A função *Map* recebe uma lista como entrada, e aplicando uma função dada, gera uma nova lista como saída. Um exemplo simples é aplicar um fator multiplicador a uma lista, por exemplo, dobrando o valor de cada elemento:

```
map({1,2,3,4}, (x2)) > {2,4,6,8}
```

Nesse exemplo, para a lista de entrada {1,2,3,4} foi aplicado o fator multiplicador 2, gerando a lista {2,4,6,8}. Podemos verificar que a função é aplicada a todos os elementos da lista de entrada. Logo, cada iteração na lista de entrada vai gerar um elemento da lista de saída. A função de mapeamento no exemplo dado poderia se chamar "dobro". A chamada com a função dobro pode ser expressa como:

```
map({1,2,3,4}, dobro) > {2,4,6,8}
```

A função *Reduce*, similarmente à função *Map*, vai receber como entrada uma lista e, em geral, aplicará uma função para que a entrada seja reduzida a um único valor na saída. Algumas funções do tipo *Reduce* mais comuns seriam "mínimo", "máximo" e "média". Aplicando essas funções ao exemplo temos as seguintes saídas:

```
reduce({2,4,6,8}, mínimo) > 2
reduce({2,4,6,8}, máximo) > 8
reduce({2,4,6,8}, média) > 5
```

No paradigma MapReduce, as funções *Map* e *Reduce* são utilizadas em conjunto e, normalmente, as saídas produzidas pela execução das funções *Map* são utilizadas como entrada para as funções *Reduce*. Associando as funções dos exemplos apresentados, podemos expressar o seguinte conjunto de funções aninhadas:

```
reduce(map({1,2,3,4}, dobro), mínimo) > 2
reduce(map({1,2,3,4}, dobro), máximo) > 8
reduce(map({1,2,3,4}, dobro), média) > 5
```

Google MapReduce

O paradigma de programação MapReduce demonstrou ser adequado para trabalhar com problemas que podem ser particionados ou fragmentados em subproblemas. Isso porque podemos aplicar separadamente as funções *Map* e *Reduce* a um conjunto de dados. Se os dados forem suficientemente grandes, podem ainda ser particionados para a execução de diversas funções *Map* ao mesmo tempo, em paralelo. Essas características despertaram a atenção ao paradigma, que entrou em evidência novamente quando foi implementado pela Google, utilizando os conceitos de programação paralela e distribuído, conforme o histórico apresentado na Seção 3.1.1. Duas contribuições principais do Google MapReduce se destacam:

- As funções *Map* e *Reduce* deixaram de ser restritas ao paradigma de programação funcional sendo disponibilizadas em bibliotecas Java, C++ e Python;
- O MapReduce foi introduzido na computação paralela e distribuída. Isso foi feito, pela explícita retroalimentação dos resultados da função *Map* como entrada para função *Reduce*, conforme os exemplos anteriores. A abordagem permite que os dados distribuídos ao longo dos nós de um aglomerado sejam utilizados nas funções *Map* e *Reduce* quando necessários.

Nessa implementação, a ideia ainda permanece a mesma: aplicar uma função *Map* em um conjunto de valores e utilizar a saída produzida para aplicar a função *Reduce*, gerando a saída final da computação. A abordagem adota o princípio de abstrair toda a complexidade da paralelização de uma aplicação usando apenas as funções *Map* e *Reduce*. Embora o paradigma MapReduce fosse conhecido, a implementação da Google deu sobrevida ao mesmo. A ideia simples das funções demonstrou ser um método eficaz de resolução de problemas usando programação paralela, uma vez que tanto *Map* quanto *Reduce* são funções sem estado associado e, portanto, facilmente paralelizáveis. Na Figura 3.5 é apresentado o modelo MapReduce desenvolvido pela Google para ser aplicado em aglomerados. Grande parte do sucesso dessa recriação do MapReduce foi alavancada pelo desenvolvimento do sistema de arquivos distribuído Google File System, ou simplesmente GFS, cujos conceitos foram aproveitados para gerar as versões preliminares do arcabouço Apache Hadoop.

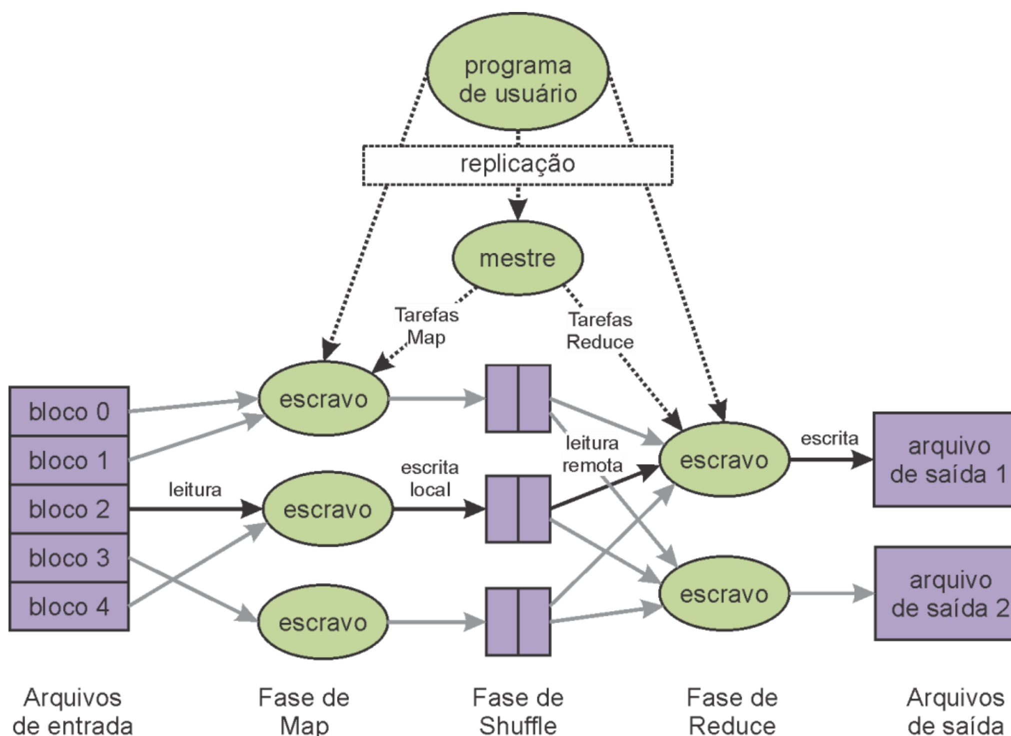

Figura 3.5. Modelo MapReduce implementado pela Google (Adaptado de: Dean & Ghemawat, 2008)

O Hadoop MapReduce pode ser visto como um paradigma de programação que expressa computação distribuída como uma sequência de operações distribuídas em conjuntos de dados. Para tal, a base de uma aplicação MapReduce consiste em dividir e processar esses dados, com o uso das funções *Map* e *Reduce*. As funções *Map* utilizam

os blocos dos arquivos armazenados com entrada. Os blocos podem ser processados em paralelo em diversas máquinas do aglomerado. Como saída, as funções *Map* produzem, normalmente, pares chave/valor. As funções *Reduce* são responsáveis por fornecer o resultado final da execução de uma aplicação, juntando os resultados produzidos por funções *Map*. Essa composição denota claramente como o Apache Hadoop tomou proveito das melhores características do Google MapReduce.

Quando aplicado ao ambiente distribuído, como em um aglomerado, o Hadoop MapReduce executa um conjunto de funções *Map* e *Reduce* definidas pelo usuário. Essas funções são denominadas tarefa pelo Hadoop. A computação é distribuída e controlada pelo arcabouço, que utiliza o seu sistema de arquivos (HDFS) e os protocolos de comunicação e troca de mensagens, para executar uma aplicação MapReduce. O processamento tem três fases: uma fase inicial de mapeamento, onde são executadas diversas tarefas *Map*; uma fase intermediária onde os dados são recolhidos das funções *Map*, agrupados e disponibilizados para as tarefas de *Reduce*; e uma fase de redução onde são executadas diversas tarefas *Reduce*, para agrupar os valores comuns e gerar a saída da aplicação.

Os dados utilizados na fase de mapeamento, em geral, devem estar armazenados no HDFS. Dessa forma os arquivos contendo os dados serão divididos em um número de blocos e armazenados no sistema de arquivos. Cada um desses blocos é atribuído a uma tarefa *Map*. A distribuição das tarefas *Map* é feita por um escalonador que escolhe quais máquinas executarão as tarefas. Isso permite que o Hadoop consiga utilizar praticamente todos os nós do aglomerado para realizar o processamento. Ao criar uma função *Map*, o usuário deve declarar quais dados contidos nas entradas serão utilizados como chaves e valores. Ao ser executada, cada tarefa *Map* processa pares de chave/valor. Após o processamento, a tarefa produz um conjunto intermediário de pares chave/valor. De maneira mais genérica, para cada par de chave/valor (k1, v1), a tarefa *Map* invoca um processamento definido pelo usuário, que transforma a entrada em um par chave/valor diferente (k2, v2). Após a execução das tarefas *Map*, os conjuntos que possuem a mesma chave poderão ser agrupados em uma lista. A geração dessa lista ocorre com a execução de uma função de combinação, opcional, que agrupa os elementos para que a fase intermediária seja realizada de maneira mais eficiente. De maneira genérica temos:

```
map(k1,v1) → list(k2,v2) (I)
```

Após o término das execuções das tarefas de *Map* o arcabouço executa uma fase intermediária denominada *Shuffle*. Essa fase agrupa os dados intermediários pela chave e produz um conjunto de tuplas (k2, list(v2)). Assim todos os valores associados a uma determinada chave serão agrupados em uma lista. Após essa fase intermediária, o arcabouço também se encarrega de dividir e replicar os conjuntos de tuplas para as tarefas *Reduce* que serão executadas. A fase de *Shuffle* é a que mais realiza troca de dados (E/S), pois os dados de diversos nós são transferidos entre si para a realização das tarefas de *Reduce*.

Na fase de redução, cada tarefa consome o conjunto de tuplas (k2, lista(v2)) atribuído a ele. Para cada tupla, uma função definida pelo usuário é chamada e transformando-a em uma saída formada por uma lista de pares chave/valor (k3, v3).

Novamente, o arcabouço se encarrega de distribuir as tarefas e fragmentos pelos nós do aglomerado. Esse conjunto de ações também pode ser expresso da seguinte forma:

```
reduce(k2,list(v2)) → list(k3,v3) (II)
```

Um exemplo didático de aplicação

A descrição do paradigma pode ser ilustrada de maneira mais clara usando o clássico exemplo de contar palavras (WordCount), que ilustra de maneira pedagógica a execução de uma aplicação MapReduce usando Hadoop. O arquivo **`WordCount.java`** contém o exemplo, e é distribuído como parte do pacote de exemplos do arcabouço Apache Hadoop. Esse exemplo pode ser utilizado em aplicações como análise de arquivos de registros, e tem como entrada de dados um conjunto de arquivos texto a partir dos quais a frequência das palavras será contada. Como saída, será gerado um arquivo texto contendo cada palavra e a quantidade de vezes que foi encontrada nos arquivos de entrada. Para tal, vamos imaginar dois arquivos texto distintos. O arquivo "**`entrada1.txt`**" conterá as frases: "CSBC JAI 2012" e "CSBC 2012 em Curitiba". O arquivo "**`entrada2.txt`**" conterá as palavras "Minicurso Hadoop JAI 2012", "CSBC 2012 Curitiba Paraná". Ambos os arquivos serão utilizados para ilustrar o funcionamento do exemplo. Ao final do exemplo será apresentado o código-fonte da aplicação para alguns destaques.

Para executar o exemplo no Hadoop, após ter seus processos iniciados, em um terminal, estando no diretório de instalação do arcabouço, podemos executar a seguinte linha de comando:

```
bin/hadoop jar hadoop-*-examples.jar wordcount entradas/ saídas/
```

Estamos assumindo que o Hadoop está em execução no modo local e, portanto o diretório "**`entradas`**" deve estar criado e conter os arquivos "**`entrada1.txt`**" e "**`entrada2.txt`**". O arcabouço é chamado e como parâmetro recebe um arquivo JAR. Dentro do arquivo, a classe **`WordCount`** foi selecionada para execução, que levará em consideração todos os arquivos dentro do diretório "**`entradas`**". Ao invés de um diretório, um único arquivo também pode ser passado como parâmetro de entrada. Ao invés do nome do diretório basta especificar o caminho completo para o arquivo. O arquivo produzido como resultado estará no diretório "**`saídas`**", que será automaticamente criado caso ainda não exista. Se o Hadoop estiver em execução no modo pseudo-distribuído ou em um aglomerado, os arquivos de entrada devem ser enviados ao HDFS. Os comandos a seguir ilustram essas operações. Neles os arquivos são enviados ao HDFS para um diretório chamado "**`entradas`**" que, caso não exista, será criado.

```
bin/hadoop fs –put ~/entrada1.txt entradas/entrada1.txt

bin/hadoop fs –put ~/entrada2.txt entradas/entrada2.txt
```

Uma vez em execução no Hadoop, a aplicação será dividida em duas fases, conforme o paradigma apresentado: a fase de mapeamento e a fase de redução. Na fase de mapeamento, cada bloco dos arquivos texto será analisado individualmente em uma função *Map* e para cada palavra encontrada será gerada uma tupla contendo o par

chave/valor contendo a palavra encontrada e o valor 1. Ao final da fase de mapeamento a quantidade de pares chave/valor gerados será igual ao número de palavras existentes nos arquivos de entrada. Aplicada ao exemplo, a fase de mapeamento produziria a saída apresentada no Quadro 3.14.

Quadro 3.14. Saídas produzidas pela fase de mapeamento

Arquivo entrada1.txt:	Arquivo entrada2.txt:
(CSBC, 1)	(Minicurso, 1)
(JAI, 1)	(Hadoop, 1)
(2012, 1)	(JAI, 1)
(CSBC, 1)	(2012, 1)
(2012, 1)	(CSBC, 1)
(em, 1)	(2012, 1)
(Curitiba, 1)	(Curitiba, 1)
	(Paraná, 1)

Ao final da execução da fase de mapeamento, a fase intermediária é executada com a função de agrupar os valores de chaves iguais em uma lista, produzindo a saída apresentada no Quadro 3.15.

Quadro 3.15. Saídas produzidas após a execução da fase intermediária

```
(2012, [2, 2])
(CSBC, [2, 1])
(Curitiba, [1, 1])
(em, 1)
(Hadoop, 1)
(JAI, [1, 1])
(Minicurso, 1)
(Paraná, 1)
```

Em seguida, os dados serão disponibilizados para a fase de redução. Serão executadas tarefas *Reduce* com o intuito de reduzir valores de chaves iguais provenientes de diferentes execuções de tarefas *Map*. Após a execução das tarefas *Reduce* a seguinte saída será gerada:

Quadro 3.16. Saídas produzidas pela fase de redução

```
(2012, 4)
(CSBC, 3)
(Curitiba, 2)
(em, 1)
(JAI, 2)
(Hadoop, 1)
(Minicurso, 1)
(Paraná, 1)
```

O exemplo ilustra as etapas intermediárias e as saídas geradas pela execução de uma aplicação MapReduce no Hadoop. O arquivo **`WordCount.java`** contém a classe **`TokenizerMapper`** responsável pela função de mapeamento. O Quadro 3.17 apresenta o código-fonte da classe escrito em Java.

A classe abstrata **`Mapper`**[16] é um tipo genérico, com quatro parâmetros formais de tipo. Estes parâmetros especificam os tipos dos valores esperados para os pares chave/valor de entrada e saída (**`Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`**). No

[16] http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapreduce/Mapper.html

Quadro 3.17 podemos ver como a classe `TokenizerMapper` estende a classe `Mapper` (`Mapper<Object, Text, Text, IntWritable>`). Podemos deduzir que a classe considera como chave de entrada objetos (arquivos) que possuem como valores texto (palavras). Como saída produzirá chaves do tipo texto (palavras) que possuem como valores inteiros (quantidade).

Quadro 3.17. Classe TokenizerMapper

```
 1. public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{
 2.   private final static IntWritable one = new IntWritable(1);
 3.   private Text word = new Text();
 4.   public void map(Object key, Text value, Context context
                 ) throws IOException, InterruptedException {
 5.     StringTokenizer itr = new StringTokenizer(value.toString());
 6.     while (itr.hasMoreTokens()) {
 7.       word.set(itr.nextToken());
 8.       context.write(word, one);
 9.     }
10.   }
11. }
```

Dentro da classe o método `map` que contém a assinatura `map(KEYIN key, VALUEIN value, Mapper.Context context)`, foi implementado como `map(Object key, Text value, Context context)` e será executado uma vez para cada par chave/valor da entrada. Nesse caso a entrada são arquivos texto, que serão processados e para cada palavra dos arquivos será emitida uma tupla do tipo `(palavra, 1)`.

A classe `IntSumReducer`, responsável pelo código da fase de *Reduce*, contém o código apresentado no Quadro 3.18. De maneira semelhante, a classe abstrata `Reducer`[17] também é um tipo genérico definido como `Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`, onde também são especificados pares de chaves/valores tanto para entrada como saída. Na implementação do exemplo podemos verificar que o método `IntSumReducer` estende a classe `Reducer` (`Reducer<Text, IntWritable, Text, IntWritable>`). Isso indica que indica que a classe receberá como entrada pares chave/valor do tipo texto (palavras) / inteiros (quantidade) e que produzirá saídas do mesmo tipo: texto (palavras) / inteiros (quantidade).

O método `reduce`, do Quadro 3.18, implementado como `reduce(Text key, Iterable<IntWritable> values, Context context)`, será chamado para cada par chave/(coleção de valores) das entradas ordenadas. O método itera nas entradas para realizar a soma das quantidades de cada palavra encontrada nos arquivos emitindo como saída tuplas do tipo `(palavra, quantidade)`.

Deve ser ressaltado que a partir da versão 0.20.x, o Hadoop transformou as interfaces `Mapper` e `Reducer` em classes abstratas. Uma nova API foi desenvolvida de maneira a facilitar as implementações de MapReduce. Usando a nova API podemos adicionar um método a uma classe abstrata sem quebrar implementações anteriores da classe. A modificação já pode ser observada nas duas classes apresentadas

[17] http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapreduce/Reducer.html

TokenizerMapper e **IntSumReducer** que estendem respectivamente as classes abstratas **Mapper** e **Reducer**.

Quadro 3.18. Classe IntSumReducer

```
 1. public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
 2.  private IntWritable result = new IntWritable();
 3.  public void reduce(Text key, Iterable<IntWritable> values,
                      Context context
                      ) throws IOException, InterruptedException {
 4.    int sum = 0;
 5.    for (IntWritable val : values) {
 6.      sum += val.get();
 7.    }
 8.    result.set(sum);
 9.    context.write(key, result);
10.  }
11. }
```

Quadro 3.19. Método principal da classe WordCount

```
 1. public static void main(String[] args) throws Exception {
 2.    Configuration conf = new Configuration();
 3.    String[] otherArgs =
           new GenericOptionsParser(conf, args).getRemainingArgs();
 4.    if (otherArgs.length != 2) {
 5.      System.err.println("Usage: wordcount <in> <out>");
 6.      System.exit(2);
 7.    }
 8.    Job job = new Job(conf, "word count");
 9.    job.setJarByClass(WordCount.class);
10.    job.setMapperClass(TokenizerMapper.class);
11.    job.setCombinerClass(IntSumReducer.class);
12.    job.setReducerClass(IntSumReducer.class);
13.    job.setOutputKeyClass(Text.class);
14.    job.setOutputValueClass(IntWritable.class);
15.    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
16.    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
17.    System.exit(job.waitForCompletion(true) ? 0 : 1);
18.  }
```

Até agora foram citados pares de chaves/valores sem falar especificamente dos tipos de dados. Note que existe uma correlação entre os tipos de dados primitivos das linguagens de programação e o Hadoop. No exemplo ilustrado a aplicação recebe como entrada um conjunto de arquivos do tipo texto contendo cadeias de caracteres (*Strings*), e os processa de forma a obter a quantidade de vezes (um número inteiro) que uma palavra aparece no conjunto de entrada. O Hadoop definiu a maneira como os pares de chave/valor serão serializados para que possam ser distribuídos através do aglomerado.

Somente classes que utilizam esses padrões de serialização definidos pelo arcabouço podem ser chaves ou valores em uma execução (Lam, 2010).

Especificamente, as classes que implementam a interface `Writable` só podem assumir o papel de valores. Já as classes que implementam a interface `WritableComparable<T>` podem ser chaves ou valores dentro do arcabouço. Essa última é uma combinação das interfaces `Writable` e `java.lang.Comparable<T>`. Isso porque as chaves precisam de requisitos de comparação para sua ordenação na fase de redução. A distribuição do Hadoop já vem com um número razoável de classes predefinidas que implementam a interface `WritableComparable`, incluindo classes Wrappers para os principais tipos primitivos de dados do Java (ver Tabela 3.1).

Nos Quadros 3.17 e 3.18 podem ser vistos os tipos de dados utilizados no exemplo. Com exceção da chave de entrada da classe `TokenizerMapper` que é um `Object` por trabalhar com arquivos, o restante dos dados são do tipo texto ou números inteiros e, portanto, usam as classes Wrappers `Text` e `IntWritable`.

Tabela 3.1. Lista das classes Wrappers que implementam WritableComparable

Classe Wrapper	Descrição
`BooleanWritable`	Wrapper para valores boolean padrão
`ByteWritable`	Wrapper para um byte simples
`DoubleWritable`	Wrapper para valores double
`FloatWritable`	Wrapper para valores float
`IntWritable`	Wrapper para valores inteiros
`LongWritable`	Wrapper para valores inteiros longos
`Text`	Wrapper para armazenar texto (UTF8) similar à classe `java.lang.String`
`NullWritable`	Reservado para quando a chave ou valor não é necessário

Trabalhos (*jobs*) e tarefas (*tasks*) no Hadoop

No Quadro 3.19 é apresentado o código-fonte do método principal do programa `WordCount.java`. No quadro observamos que o arcabouço Hadoop trabalha com o conceito de `Job`, que doravante será referenciado como trabalho. Um trabalho pode ser considerado como uma execução completa de um programa MapReduce incluindo todas as suas fases: *Map*, *Shuffle* e *Reduce*. Para tal, é necessário instanciá-lo e definir algumas propriedades.

Um trabalho é criado ao se criar uma instância da classe `Job`, passando como parâmetros uma instância da classe `Configuration` e um identificador para o trabalho. A instância da classe `Configuration` permite que o Hadoop acesse os parâmetros de configuração (também podem ser denominados recursos) definidos nos arquivos `core-default.xml` e `core-site.xml`. A aplicação pode definir ainda recursos adicionais que serão carregados subsequentemente ao carregamento dos arquivos padrões. Uma vez instanciada a classe, as propriedades podem ser configuradas de acordo com a necessidade da execução da aplicação.

A primeira propriedade a ser ajustada é a localização do arquivo JAR da aplicação. Isso é feito pelo método `setJarByClass(Class<?> cls)`. A seguir são apontadas as classes do MapReduce que serão utilizadas durante as fases do trabalho,

que são determinadas por métodos específicos. A classe `Mapper` é configurada pelo método `setMapperClass(Class<? extends Mapper> cls)`, a classe `Combiner` pelo método `setCombinerClass(Class<? extends Reducer> cls)` e a classe `Reducer` pelo método `setReducerClass(Class<? extends Reducer> cls)`.

Em seguida são determinados os tipos de dados que serão produzidos pela aplicação. Dois métodos são utilizados: `setOutputKeyClass(Class<?> theClass)` e `setOutputValueClass(Class<?> theClass)` que determinam, respectivamente, o tipo das chaves e dos valores produzidos como saída do trabalho. São definidos também os caminhos contendo os arquivos de entrada para a aplicação e o local onde os arquivos de saída serão escritos.

O último método a ser executado de um trabalho é `waitForCompletion(boolean verbose)` que submete o trabalho e suas configurações ao aglomerado e espera pela conclusão da execução. Caso o parâmetro `verbose` seja definido como `true`, todas as etapas do trabalho serão exibidas no terminal de execução.

Embora já citado anteriormente, o fluxo lógico específico da aplicação MapReduce exemplo pode ser visto na Figura 3.6.

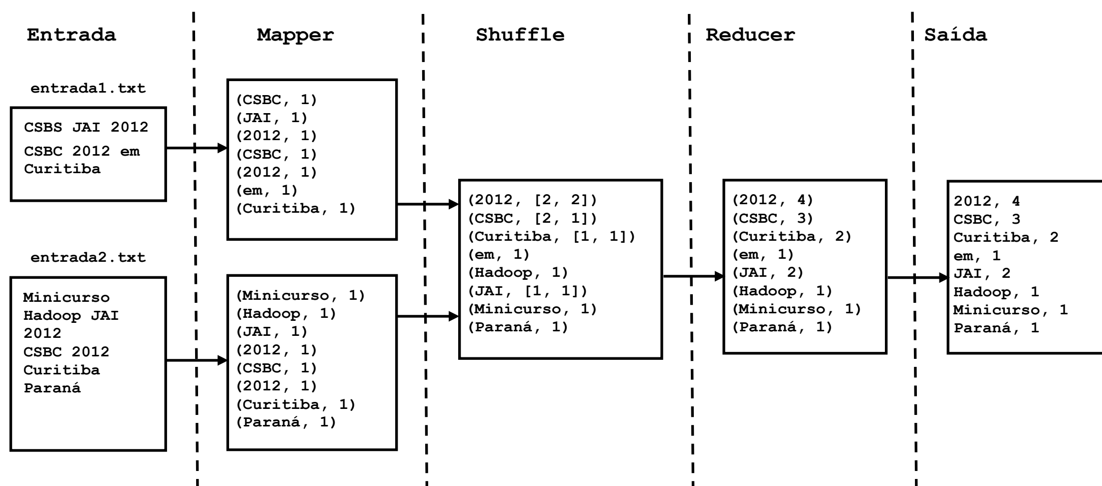

Figura 3.6. Fluxo lógico de execução da aplicação MapReduce do exemplo WordCount

O fluxo lógico de execução dessa aplicação pode ser facilmente visualizado devido ao seu baixo grau de complexidade. As fases podem ser bem definidas e divididas. Isso ocorre também porque o arcabouço esconde os detalhes de como a execução ocorre no plano físico. A intenção é tornar o Hadoop amigável ao usuário, escondendo detalhes de implementação como escalonamentos, controle de redundância e recuperação de falhas. Embora muito similar, no plano físico é possível observar o armazenamento dos dados em blocos no sistema de arquivos e a movimentação de dados e troca de mensagens de maneira a executar o trabalho. A Figura 3.7 mostra o fluxo de execução da aplicação exemplo, mas sob o ponto de vista da execução física de uma aplicação MapReduce no Hadoop.

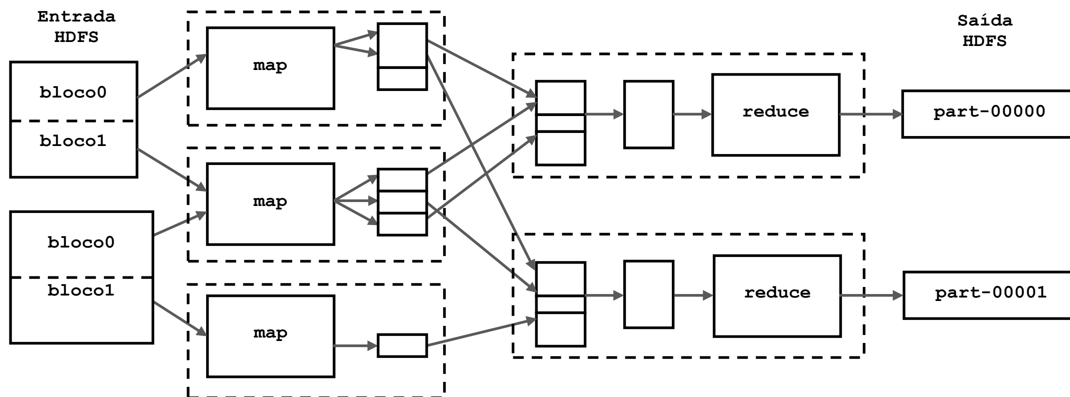

Figura 3.7. Fluxo de execução física de uma aplicação MapReduce no Hadoop

Ainda que possua modos para execução em uma única máquina, as ilustrações apresentadas nas Figuras 3.6 e 3.7 ressaltam que o arcabouço Hadoop foi desenvolvido para executar em aglomerados e grades computacionais de forma a aproveitar o melhor da computação distribuída e paralela, usando seu sistema de arquivos distribuído.

Outros exemplos

Para ilustrar o funcionamento do Apache Hadoop, Tom White (White, 2010) apresenta em seu livro, um exemplo de mineração de dados usando Hadoop. O conjunto de dados usado no exemplo foi coletado por sensores climáticos espalhados pelo planeta e podem ser aéreos, terrestres ou marítimos. Alguns dados são provenientes de satélites. Os sensores coletam e reúnem diversas informações que compõem o clima local, sua localização (latitude e longitude), altitude, direção do vento e temperatura local. O conjunto de dados utilizado é disponibilizado pelo National Climate Data Center (NCDC) em seu sítio da Internet. Basicamente existem dados climáticos sobre todos os dias do ano durante um século, de 1901 a 2001. O exemplo pretende encontrar a maior temperatura global para cada ano disponibilizado.

Cada leitura reúne as informações coletadas pelos sensores, e é processada de forma a ser armazenada em apenas uma cadeia de caracteres. O conjunto das leituras realizadas durante um ano, proveniente dos diversos sensores, é agrupado em um arquivo texto. Dentro dos arquivos, cada linha representa uma leitura de um sensor. A aplicação MapReduce desse exemplo vai retirar, de cada uma das entradas dos sensores, o ano em que a leitura foi realizada e a temperatura do ar, ambos em destaque na Figura 3.8. A reprodução do processo também pode ser vista na mesma figura. Considere a execução da aplicação para os dados dos anos de 1901 e 1902.

Outro exemplo prático é o uso do Hadoop para a geração de recomendações. Sistemas de recomendação criam sugestões personalizadas e tem se difundido amplamente no comércio eletrônico. As sugestões de produtos similares que um usuário recebe quando navegando por um sítio de compras na Internet utilizam esse tipo de tecnologia. Há mais de uma década os sistemas de recomendação fazem parte de grandes sítios de comércio eletrônico, como a Amazon, CDNow e MovieFinder.

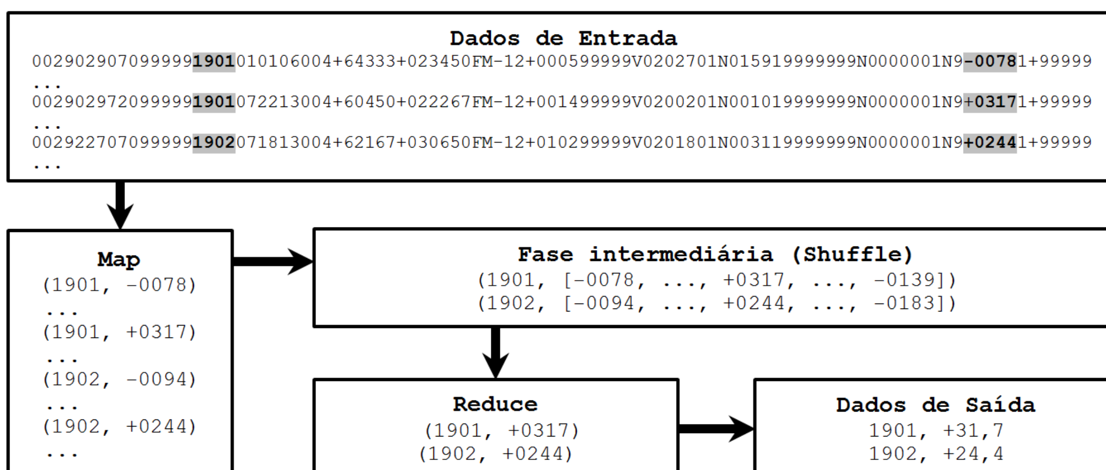

Figura 3.8. Mineração dos dados climáticos do NCDC usando Hadoop MapReduce

A Apache possui um projeto chamado Mahout, cujo principal objetivo é aproveitar o poder do MapReduce para realizar suas computações no Hadoop. Os principais algoritmos de classificação, agrupamento, e processamento em lote de filtragem colaborativa existentes são implementados e disponibilizados no projeto Mahout.

Em uma aplicação recente utilizamos o Apache Mahout para gerar recomendações de filmes para usuário usando um algoritmo de filtragem colaborativa. Nesses algoritmos, as recomendações para um usuário são resultado da comparação das avaliações ou recomendações de itens atribuídas por outros usuários. Com base em incidências comuns, é possível gerar uma recomendação para um item. Nesse exemplo, o método básico consiste em selecionar um conjunto de filmes que serão avaliados por espectadores. Cada filme visto por um espectador recebe uma avaliação, por exemplo, um valor de 1 a 5. Usando o conjunto de dados gerado por todos os espectadores, podemos atribuir valores para os filmes que uma pessoa ainda não viu, isto é, gerar recomendações de filmes que o espectador pode gostar de acordo com o que já viu e classificou anteriormente. Imagine a situação hipotética apresentada na Tabela 3.2. Para os filmes que Beltrano ainda não assistiu, o sistema deve gerar valores de recomendação. Um mecanismo de cálculo é pela similaridade nas avaliações de outros usuários, conforme destacado na tabela.

Tabela 3.2. Situação exemplo para o uso de filtragem colaborativa

Usuários	Filme 1	Filme 2	Filme 3	Filme 4
Fulano	**5**	2	**4**	**4**
Cicrano	2	2	**4**	**5**
Beltrano	**4**	???	**5**	???

Na situação apresentada a quantidade de dados é relativamente pequena. Em nosso exemplo, trabalhamos com um conjunto de dados que possui mais de 10 milhões de recomendações de mais 70 mil usuários em um conjunto de aproximadamente 10.500 filmes, o que justifica o uso do Mahout no Hadoop. Para gerar recomendações uma aplicação MapReduce é criada usando as bibliotecas do Mahout. Em sua execução no Hadoop, os dados de entrada o são o conjunto de avaliações de filmes preenchidas

pelos os usuários e os usuários para os quais se deseja gerar as recomendações. A execução da aplicação envolve um conjunto de operações com vetores e matrizes sobre um grande conjunto de dados, operações essas que podem ser paralelizadas. Assim, novamente ressaltamos a escolha do MapReduce no Hadoop para obtenção dos resultados.

A aplicação de recomendação por filtragem colaborativa é constituída por uma sequência de trabalhos MapReduce. O Mahout se encarrega de encadear os trabalhos de forma que a saída de um trabalho seja usada como entrada do próximo. Ao todo podemos considerar a execução de dez trabalhos MapReduce para gerar um conjunto de recomendações usando o `RecommenderJob` do Mahout, explicados de maneira sucinta na lista a seguir:

- Fase 1: Gera a lista de identificadores dos filmes;
- Fase 2: Cria o vetor de preferências para cada usuário;
- Fase 3: Conta os usuário de maneira única;
- Fase 4: Transpõe os vetores de preferências dos usuários;
- Fase 5: Calcula similaridade entre os vetores de preferência gerando uma matriz de similaridades;
- Fases 6, 7 e 8: Realiza preparação e multiplicação da matriz de similaridades pelos vetores de preferências dos usuários;
- Fase 9: Filtra os itens para os usuários selecionados;
- Fase 10: Emite a quantidade de recomendações desejada para cada usuário.

Quando em execução no Hadoop, essa aplicação é considerada como apenas um trabalho. Entretanto, para cada fase citada, um novo trabalho é criado. Cada trabalho é dividido em várias tarefas de *Map* e *Reduce*, paralelizando assim a computação. Esse paralelismo fica evidente quando ocorrem as fases que trabalham com geração e multiplicação de vetores e matrizes, onde partes de cada vetor ou matriz podem ser processados em diferentes nós do aglomerado. Para ilustrar a complexidade de processamento envolvida nesses algoritmos, para a geração de uma lista de 10 filmes para um conjunto de 30 usuários, o tempo gasto para gerar o conjunto de recomendações fica entre 15 e 20 minutos. O aglomerado onde foram realizados os testes possui 15 máquinas como processadores de 2 núcleos e com 2Gb de memória RAM em média. Para a realização dos testes também foram utilizados diferentes algoritmos para o cálculo de similaridade, o que justifica a diferença no tempo das execuções e precisão das recomendações. Estes exemplos ressaltam, mais uma vez, a capacidade do Hadoop em executar e controlar diversos trabalhos MapReduce ao mesmo tempo. Esse controle é realizado pelos mecanismos de escalonamento implementados no arcabouço, apresentados a seguir.

Escalonamento de tarefas

Em suas versões iniciais, o Hadoop tinha uma abordagem de escalonamento simples, usando uma estrutura tipo fila, onde os trabalhos submetidos aguardavam e eram executados de acordo com a ordem de chegada. Dessa forma, cada trabalho bloqueava o aglomerado todo durante sua execução, mesmo que não utilizasse todos os recursos ou nós disponíveis. Mais tarde, como uma evolução, foi estabelecido o mecanismo de prioridades, onde os trabalhos com maiores prioridades são movidos para o início da fila de execução. Entretanto, esse mecanismo não tinha suporte à preempção, e trabalhos

com alta prioridade ainda poderiam ficar bloqueados na fila esperando um trabalho longo com baixa prioridade terminar sua execução. O escalonador padrão do Hadoop ainda é o de fila simples com prioridade. Entretanto, isso pode ser alterado, pois a distribuição contém pelo menos outros três mecanismos disponíveis: o escalonador justo (Fair Scheduler), escalonador de capacidade (Capacity Scheduler) e o escalonador de demanda (HOD Scheduler – Hadoop on Demand Scheduler).

O escalonador justo é um método de atribuição de recursos para os trabalhos submetidos ao aglomerado de forma que todos os trabalhos obtenham, em média, uma parcela igual dos recursos ao longo do tempo. Se há apenas um trabalho em execução no aglomerado, todos os recursos disponíveis podem ser alocados para o mesmo. Quando outros trabalhos são submetidos, os recursos livres são designados para as tarefas dos novos trabalhos. Dessa forma cada trabalho submetido ao aglomerado ficará, em média, como o mesmo tempo de CPU. Essa abordagem permite que trabalhos menores terminem em tempos razoáveis e evita que trabalhos longos sofram *starvation*. O escalonador justo ainda pode trabalhar com prioridades para os trabalhos. As prioridades são usadas como pesos para determinar quanto tempo de processamento cada trabalho vai receber.

O escalonador de capacidade foi desenvolvido para executar o Hadoop MapReduce de forma compartilhada, em um aglomerado onde várias empresas diferentes possuem acesso. Foi projetado para maximizar a capacidade e a utilização do aglomerado na execução de trabalhos MapReduce. Esse escalonador permite que cada empresa com acesso ao aglomerado tenha uma garantia mínima de capacidade de execução, em outras palavras, tempo de processamento. A ideia central é que os recursos disponíveis no aglomerado sejam divididos entre várias organizações que financiam coletivamente a manutenção da infraestrutura com base em suas necessidades de processamento. Existe ainda o benefício adicional de que uma organização pode acessar qualquer excedente de capacidade que não está sendo usado por outros. Isso proporciona elasticidade com elevado custo-benefício. Em se tratando de recursos compartilhados, o escalonador fornece um conjunto rigoroso de limites para assegurar que um único trabalho ou usuário ou fila não consuma uma quantidade desproporcional de recursos no aglomerado. Além disso, o JobTracker é um recurso primordial no arcabouço e o escalonador fornece limites para tarefas e trabalhos inicializadas e/ou pendentes de um único usuário na fila de espera, tudo para garantir a igualdade de uso e a estabilidade do aglomerado.

O escalonador de demanda ou HOD é um sistema para provisionamento e gerenciamento de instâncias independentes de Hadoop MapReduce e Hadoop Distributed File System (HDFS) em um aglomerado compartilhado. HOD é uma ferramenta que permite tanto administradores quanto usuários configurar e usar de forma rápida o Hadoop. Também é uma ferramenta útil para desenvolvedores que precisam compartilhar um aglomerado físico para testar as suas próprias versões do Hadoop. O HOD usa um gerenciador de recursos para fazer a alocação de nós e neles inicializa o HDFS e executa as aplicações MapReduce quando solicitado.

Monitoramento de progresso

A execução de um trabalho MapReduce pode ser considerada como a execução em lote de um conjunto de tarefas. É importante, ao longo da execução de um trabalho, saber do andamento e progresso das tarefas intermediárias. No Hadoop, cada trabalho e suas tarefas possuem um estado, que inclui entre outras coisas, sua condição (executando, terminado com sucesso, falhou), o progresso dos *maps* e *reduces* e os contadores do trabalho. Uma tarefa em execução deve manter registros de progresso de forma a relatar o percentual que já foi concluído. Quando uma tarefa reporta progresso, um sinal é ajustado. Essa mudança de estado faz com que informações sejam enviadas ao gerenciador de tarefas TaskTracker. A atualização é enviada então ao gerenciador de trabalhos JobTracker. Nos sinais enviados ao gerenciador de trabalhos estão incluídos os estados de todas as tarefas em execução. O gerenciador de trabalhos combina todas as atualizações de progresso das tarefas e gera uma visão global de progresso dos trabalhos em execução no aglomerado, que pode ser analisada posteriormente para uma possível melhoria de desempenho do arcabouço.

Interfaces Web do Apache Hadoop

Uma vez que o Apache Hadoop foi corretamente instalado e configurado conforme exemplificado na Seção 3.2, o andamento e histórico dos trabalhos e tarefas executadas no arcabouço podem ser consultados. Os dois mecanismos mais comuns de visualização destes dados são o terminal de execução do Hadoop e a interface Web do arcabouço.

Quando a execução de uma aplicação é submetida ao Hadoop via linha de comando, todos os passos podem ser visualizados durante sua execução. Estão incluídos como saída padrão todos os resultados intermediários gerados. Podem ser vistos os andamentos das tarefas de *Map* e *Reduce* da aplicação. Se ocorrer algum erro durante a execução a exceção gerada também pode ser conferida no terminal. Embora essas informações estejam disponíveis em tempo de execução, todas elas são gravadas em arquivos de registros e disponibilizadas no arcabouço. Outro meio de acessar essas informações é por meio das interfaces Web disponíveis no Hadoop. São três interfaces disponibilizadas para acompanhamento dos históricos dos trabalhos e tarefas.

A primeira interface é a do gerenciador de trabalhos JobTracker. Essa interface é disponibilizada pela porta 50030 do arcabouço. Se rodando em modo local ou pseudo-distribuído pode ser acessada pelo endereço `http://localhost:50030/` . Se estiver em um aglomerado basta substituir o *localhost* pelo endereço IP do nó principal. A interface apresentada na Figura 3.9 fornece informações estatísticas sobre os trabalhos executados no arcabouço. São mostrados trabalhos em execução, completos e que falharam. Na interface também é possível consultar os arquivos de registro de cada um dos trabalhos executados no arcabouço.

A segunda interface Web permite visualizar o andamento das tarefas em execução no arcabouço. Podem ser vistas informações sobre tarefas dos diversos trabalhos em execução. Essa interface pode ser acessada pela porta 50060. No modo local ou pseudo-distribuído o acesso pode ser feito pelo endereço: `http://localhost:50030/` .

Quick Links
Scheduling Info
Running Jobs
Retired Jobs
Local Logs

master Hadoop Map/Reduce Administration

State: RUNNING
Started: Tue May 22 11:33:34 BRT 2012
Version: 1.0.1, r1243785
Compiled: Tue Feb 14 08:15:38 UTC 2012 by hortonfo
Identifier: 201205221133

Cluster Summary (Heap Size is 30.06 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Graylisted Nodes	Excluded Nodes
0	0	0	13	0	0	0	0	26	26	4,00	0	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Retired Jobs

Figura 3.9. Interface Web do JobTracker no Hadoop

Existe ainda uma interface que permite a visualização do sistema de arquivos HDFS, ilustrada na Figura 3.10. Esse acesso é feito pelo processo NameNode. Podemos ter um resumo do aglomerado contendo o número de nós vivos, a capacidade total de armazenamento do sistema de arquivos e acesso aos arquivos armazenados no HDFS. Também é possível visualizar os arquivos com registros das operações efetuadas. O acesso é feito pela porta 50070 no endereço onde o Hadoop está disponibilizado.

NameNode 'master:9000'

Started:	Tue May 22 11:33:32 BRT 2012
Version:	1.0.1, r1243785
Compiled:	Tue Feb 14 08:15:38 UTC 2012 by hortonfo
Upgrades:	There are no upgrades in progress.

Browse the filesystem
Namenode Logs

Cluster Summary

111 files and directories, 47 blocks = 158 total. Heap Size is 46.88 MB / 888.94 MB (5%)

Configured Capacity	:	2.89 TB
DFS Used	:	100.48 MB
Non DFS Used	:	211.26 GB
DFS Remaining	:	2.68 TB
DFS Used%	:	0 %
DFS Remaining%	:	92.85 %
Live Nodes	:	13
Dead Nodes	:	0
Decommissioning Nodes	:	0
Number of Under-Replicated Blocks	:	0

Figura 3.10. Interface Web do NameNode no Hadoop

Mecanismos de tolerância a falhas

Para que todo o potencial de paralelismo do arcabouço Apache Hadoop possa ser aproveitado, o modo de execução completamente distribuído é o recomendado. Entretanto, devemos ressaltar que por se tratar de um conjunto de máquinas trabalhando em paralelo, quanto maior o número de máquinas, maior a probabilidade de ocorrerem falhas. Estamos trabalhando com máquinas que estão sujeitas a falhas em seus componentes de hardware, em seus processos do sistema operacional ou até mesmo na execução de código defeituoso. Quando um trabalho MapReduce é criado e inicia sua execução, as tarefas pertencentes ao trabalho são distribuídas ao longo dos nós do aglomerado. Caso alguma das tarefas falhe, em qualquer fase, o Hadoop possui a habilidade de se recuperar e permitir que o trabalho seja concluído.

No caso de tarefas de um trabalho, as falhas podem ocorrer em decorrência de diversos fatores, como por exemplo, defeitos na JVM, exceções não tratadas no código, ou na comunicação de um nó. No caso de exceções não tratadas e defeitos inesperados na JVM, o gerenciador de tarefas marca a tentativa de execução como falha e libera seu espaço para outra tarefa.

Da mesma forma, se o gerenciador de tarefas não receber comunicação de progresso de um nó por um determinado período de tempo, o processo de execução da tarefa será extinto e a execução marcada como falha naquele nó. Quando ocorre uma falha em uma tarefa, o TaskTracker comunica o gerenciador de trabalhos. Assim, a execução da tarefa poderá ser escalonada novamente. O JobTracker tentará evitar que a tarefa seja escalonada para execução no mesmo nó onde ocorreu a falha.

Por padrão, uma tarefa não receberá mais tentativas de execução se sofrer um número configurado de falhas. Se ocorrerem suficientes falhas de uma mesma tarefa, isso poderá acarretar no cancelamento da execução do trabalho. Entretanto, quando apenas uma tarefa falha durante um trabalho, isso não pode implicar no cancelamento completo do trabalho em alguns casos. Para evitar essa condição, existe uma propriedade que determina o percentual de tarefas de um trabalho que podem falhar. Logo, até que esse percentual seja atingido, o trabalho não será cancelado.

Outro tipo de falha podem ser os defeitos de hardware em nós escravo do aglomerado. Se um nó sofre travamento ou está executando suas tarefas de maneira muito lenta, a comunicação com o gerenciador de tarefas vai cessar. Quando o tempo limite de espera é atingido o nó é removido da fila e não receberá tarefas para executar até ter sua situação normalizada. Se a normalização não ocorrer, o nó pode ir para uma lista negra de nós. Ademais, o nó também pode entrar nessa lista se o número de tarefas em execução que falharam é maior que a média do restante dos nós do aglomerado. Para que um nó escravo seja retirado da lista, basta que seja reiniciado e sua capacidade volte ao normal.

A falha mais grave que pode ocorrer em um aglomerado executando Hadoop é a do nó mestre, que executa o processo JobTracker. Por se tratar de uma falha severa no controlador principal do arcabouço, se não existirem réplicas, o Hadoop não consegue tratar esse tipo de falha e ficará indisponível enquanto o problema persistir. Dependendo do nível de defeito ocorrido, os trabalhos submetidos e em execução no aglomerado não poderão ser recuperados e terão de ser submetidos novamente. Esse problema é alvo de alguns estudos atualmente, que tentam recuperar os dados de trabalhos em execução no Hadoop quando ocorrem falhas desse tipo. Uma abordagem, ainda em estudo, é a

replicação de metadados dos trabalhos e tarefas em nós específicos do aglomerado que podem assumir o papel do nó mestre em caso de falha.

3.5. Anatomia de um programa Hadoop

Com o mecanismo da execução de uma aplicação MapReduce no Hadoop explicado na seção anterior, esta seção vai apresentar os detalhes internos da execução de uma aplicação no Hadoop. Para entender melhor os trabalhos executados no Hadoop é necessário compreender mais a anatomia dessa execução. A Figura 3.11 mostra como o Hadoop controla a execução de seus trabalhos.

Tom White (White, 2010) cita que o processo de execução possui quatro entidades no nível mais alto:

1. O cliente, que submete o trabalho MapReduce;
2. O JobTracker ou gerenciador de trabalhos, entidade que controla a execução dos trabalhos. O JobTracker é um processo em execução no nó mestre do aglomerado e controla as execuções de trabalhos MapReduce no Hadoop;
3. O TaskTracker ou gerenciador de tarefas, controla a execução de tarefas dentro de um trabalho. O TaskTracker é um processo que em execução nos nós escravos do aglomerado e controla as execuções de tarefas *Map* ou *Reduce* nos mesmos;
4. O sistema de arquivos distribuídos (Hadoop Distributed File System), utilizado para compartilhar arquivos entre as entidades envolvidas.

Por convenção, o nó principal de um aglomerado (mestre) é a entidade NameNode do sistema de arquivos distribuídos, responsável por controlar a localização dos dados e sua replicação. Em geral, o nó mestre do aglomerado é aquele que acumula também o papel de gerenciador de trabalhos, controlando as submissões e escalonamentos dos trabalhos. Os trabalhos submetidos são divididos em tarefas pelo JobTracker que também as distribui para os nós escravos do aglomerado. Os nós escravos também são conhecidos como DataNodes, pois desempenham o papel de armazenamento dos blocos dos arquivos no aglomerado. Também acumulam o papel de TaskTrackers, e são responsáveis pela execução e controle das tarefas a eles designadas.

Para que um novo trabalho submetido ao Hadoop possa ser executado, as seguintes ações são realizadas antes do início da execução, ainda segundo (White, 2010):

- Um novo identificador (ID) é solicitado para o novo trabalho;
- A especificação do diretório de saída dos dados do trabalho é verificada. Se o diretório não foi especificado ou se já existe, o trabalho não é executado e um erro é gerado pelo arcabouço;
- O número de tarefas em que o trabalho será dividido é calculado. Se o cálculo não puder ser feito, porque o caminho de entrada dos dados não foi especificado, por exemplo, o trabalho também não é executado e um erro é gerado pelo arcabouço;
- Os recursos necessários para a execução das tarefas são copiados para o sistema de arquivos do gerenciador de trabalhos, em um diretório criado com o ID do trabalho. A cópia inclui o arquivo JAR do programa e os arquivos de configuração. Isso é feito com cópias replicadas para que os gerenciadores de

tarefas possam acessar e copiar estes dados quando receberem a designação de uma tarefa.

Em seguida o gerenciador de trabalhos é informado que o trabalho está pronto para ser executado. Recebendo essa informação, o gerenciador coloca o trabalho em uma fila de execução onde o escalonador faz suas operações. Como o trabalho é dividido em tarefas, é função do escalonador recorrer ao sistema de arquivos, por meio do NameNode, para saber onde estão os blocos dos arquivos que serão utilizados durante a execução das tarefas. As tarefas de *Map* e *Reduce* são criadas de acordo com as configurações do trabalho e seu andamento. Cada uma das tarefas criadas recebe um identificador utilizado para o controle de sua execução. As tarefas são então designadas para a execução em um nó escravo pelo escalonador.

Quando são concluídas as execuções das tarefas, os gerenciadores de tarefa responsáveis enviam os resultados de suas execuções ao gerenciador de trabalhos, encarregado de coordenar o processo. As tarefas de *Reduce* são iniciadas tão logo existam dados provenientes de tarefas *Map* disponíveis para seu início. Ao final da execução de todas as tarefas, o resultado do trabalho estará disponível nos arquivos criados no diretório de saída do trabalho. A Figura 3.11 apresenta a sequência de ações do arcabouço para execução de um trabalho MapReduce.

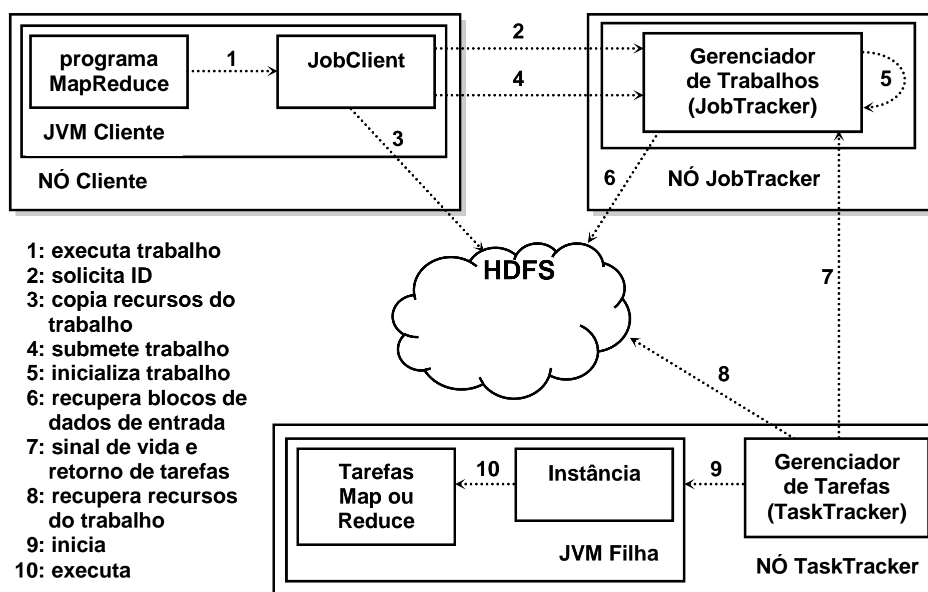

Figura 3.11. Modelo de execução de um trabalho MapReduce no Hadoop (Adaptado de: White, 2010)

No passo 6, apresentado na Figura 3.11, são recuperados os dados que serão utilizados pela tarefas do trabalho. Nesse passo pode ser determinada a quantidade de tarefas *Map* que serão criadas no trabalho. Em geral, para cada bloco dos arquivos de entrada localizado no HDFS uma tarefa *Map* é criada. O número de tarefas *Reduce* pode ser definido por meio de propriedade específica na configuração do trabalho e independe da quantidade de blocos dos arquivos de entrada. Quanto menor o número de tarefas *Reduce* em um trabalho, menor a quantidade de arquivos gerados ao final da execução do trabalho. Deve ser considerado também, que quanto menor a quantidade de

tarefas *Reduce* em um trabalho, maior será o processamento destinado para essa fase, uma vez que os dados serão agrupados (reduzidos) a um número pequeno de saídas.

Prioridade e localidade de tarefas

Cada nó gerenciador de tarefas possui um número fixo de vagas para a execução de tarefas *Map* ou *Reduce*. Por exemplo, um nó pode ser capaz de executar, simultaneamente, duas tarefas de *Map* e duas de *Reduce*. Esse número é diretamente ligado à capacidade de hardware do nó, dependendo do número de núcleos do processador e quantidade de memória RAM disponível. A prioridade de preenchimento das vagas disponíveis é maior para as tarefas *Map*. Os *slots* para essas tarefas são preenchidos primeiro e em seguida, serão alocadas tarefas de *Reduce*. Essa prioridade é determinada porque o número de tarefas *Map* em um trabalho MapReduce tende a ser maior que o número de tarefas *Reduce*. Em geral, esse número tende a ser muito maior, uma vez que para cada bloco de cada arquivo utilizado como entrada para o trabalho terá uma tarefa de *Map*, ao passo que o número de tarefas *Reduce* pode ser um determinado pelo programador.

Para designar as tarefas aos nós, o gerenciador de trabalhos leva em consideração o princípio da localidade de dados. A premissa é adotada principalmente para as tarefas *Map*. O princípio da localidade é totalmente dependente da maneira como a topologia de rede do aglomerado foi projetada e montada e de como o Hadoop foi configurado. Em uma estrutura normal de centro de dados, existem entre 30 e 40 nós em um armário, ligados por uma conexão de 1GB. Se existir mais de um armário no centro de dados, eles regularmente são interligados com conexões de 1GB ou superiores. Uma topologia de dois níveis, típica de um aglomerado Hadoop, é representada na Figura 3.12.

Para evitar tráfego de dados na rede, as tarefas *Map* são designadas para execução em nós que estejam o mais próximo possível do bloco que vai ser utilizado durante a execução da tarefa. Embora a estrutura da rede não seja mapeada em forma de árvore, é comum a existência de pelo menos dois níveis: o armário e o nó do aglomerado. Segundo Tom White (White, 2010), a ideia é que a largura de banda disponível para cada um dos seguintes cenários seja progressivamente menor:

1. Tarefas no mesmo nó;
2. Nós diferentes no mesmo armário;
3. Nós em diferentes armários do aglomerado.

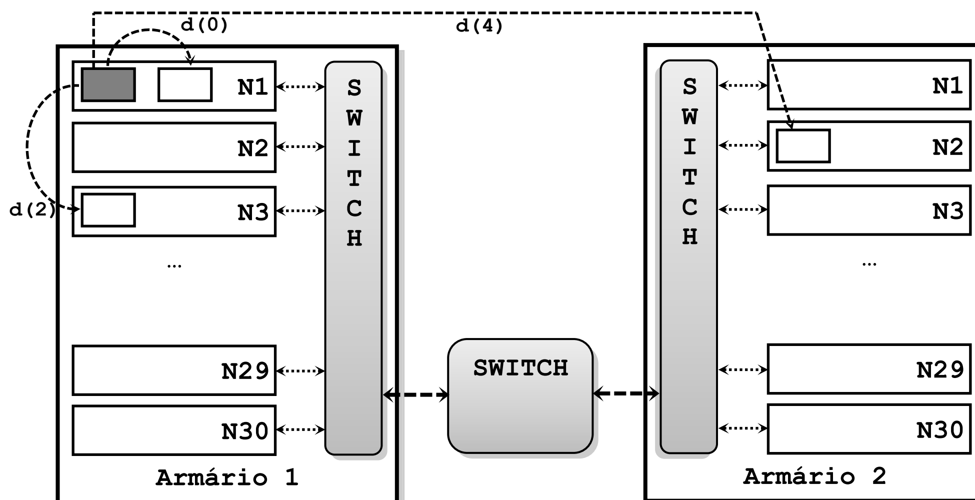

Figura 3.12. Topologia de rede de um típico aglomerado executando Hadoop

Assumindo uma notação hipotética, imagine que um nó *n1* em um armário *a1*. Podemos representar essa informação como */a1/n1*. Com essa notação podemos representar a distância entre os nós dos três cenários ilustrados:

1. *distância*(*/a1/n1*, */a1/n1*) = 0 (tarefas no mesmo nó);
2. *distância*(*/a1/n1*, */a1/n3*) = 2 (nós diferentes no mesmo armário);
3. *distância*(*/a1/n1*, */a2/n2*) = 4 (nós em diferentes armários).

Na Figura 3.12 também está representa graficamente, por uma seta tracejada, as distâncias *d(0)*, *d(2)* e *d(4)* entre as tarefas em execução em um aglomerado. Logo, com as distâncias corretamente configuradas, o Hadoop sempre espera o caso ótimo, ou seja, a menor distância entre dados e tarefas. Assim, espera-se que as tarefas de *Map* sejam executadas em nós que contenham os blocos para executar a tarefa. Entretanto, se isso não puder ocorrer, as tarefas serão alocadas para os nós com vagas disponíveis. O mesmo caso não precisa ocorrer necessariamente com as tarefas de *Reduce*. O gerenciador de trabalhos não leva em consideração o princípio da localidade para essas tarefas. Isso porque durante a fase de *Shuffle*, os resultados das tarefas *Map* são distribuídos aos nós que executarão tarefas *Reduce*. Num caso ótimo, uma tarefa *Reduce* poderia executar com os dados disponíveis das tarefas *Map* executadas no mesmo nó. Raramente esse é o caso, pois as tarefas de *Reduce* tendem a agregar os resultados provenientes de diferentes tarefas *Map* executadas em nós espalhados pelo aglomerado.

Fases intermediárias

A execução de uma aplicação MapReduce no Hadoop é citada, na maioria dos casos, com um conjunto de execuções de tarefas *Map* e *Reduce*, com os resultados das tarefas *Map* sendo alimentados como entrada nas tarefas *Reduce*. O Hadoop garante que todas as entradas de cada tarefa *Reduce* estão ordenadas por chave. Logo, nessa simplificação de processo, duas fases importantes são omitidas: a fase de *Shuffle* (embaralhamento) e a fase de *Sort* (ordenação).

A fase de *Shuffle* pode ser considerada como o ponto crucial na execução de uma aplicação MapReduce. Do lado das tarefas *Map*, quando uma tarefa *Map* começa a produzir resultados, os dados são primeiramente escritos em um buffer circular em

memória. Isso ocorre por alguns motivos. Primeiramente porque os resultados de uma tarefa *Map* podem ser divididos para uso em mais de uma tarefa *Reduce*. Em segundo porque o custo de ordenação destes dados enquanto ainda estão em memória é bem menor do que se já estivessem no disco rígido. E finalmente porque o Hadoop consegue economizar tempo de E/S escrevendo uma quantidade maior de dados em uma única operação. Alguns parâmetros de configuração podem ser configurados para obtenção de melhor desempenho por parte do arcabouço como, por exemplo, o tamanho do buffer na memória (`io.sort.mb`) e o percentual de preenchimento do buffer (`io.sort.spill.percent`) que vai disparar a operação de escrita no disco.

Antes de escrever seus resultados no disco rígido, as tarefas de *Map* dividem seus dados de saída em partições de acordo com as tarefas de *Reduce* para as quais os dados serão enviados, conforme a Figura 3.13. Ainda na memória, os dados são ordenados de acordo com a chave, para depois serem escritos no disco. Toda vez que o percentual limite do buffer é atingido, uma operação de escrita no disco é iniciada. Se nesse intervalo o buffer ficar cheio a tarefa é suspensa até que todo o conteúdo seja escrito no disco. Os dados são então disponibilizado para as tarefas *Reduce*. Para reduzir a quantidade de tráfego de dados entre os nós durante a fase de *Shuffle*, as saídas das tarefas *Map* podem ser comprimidas usando ferramentas disponíveis no sistema operacional como *gzip* e *bzip2*.

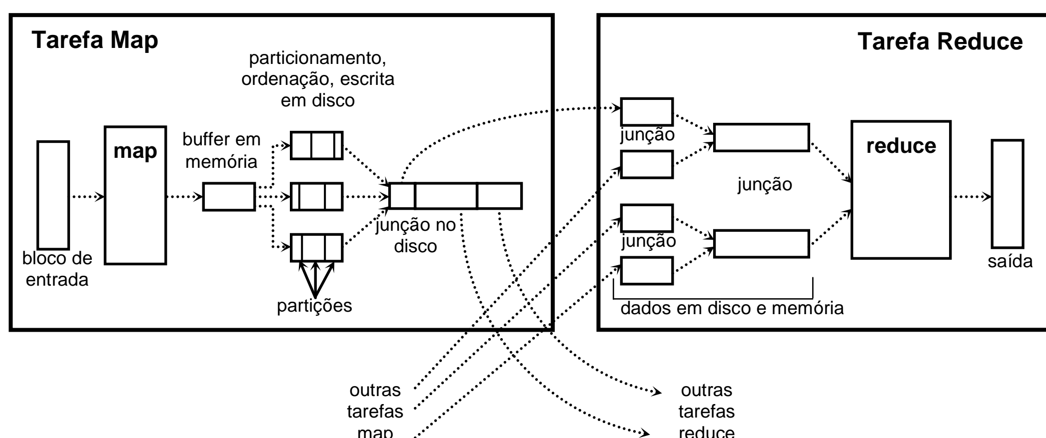

Figura 3.13. Ilustração das fases de *Shuffle* e *Sort* no Hadoop (Adaptado de: White, 2010)

Ainda na fase de *Shuffle*, pelo lado das tarefas *Reduce*, as saídas das tarefas *Map* do trabalho em execução estão nos discos dos nós onde as tarefas foram concluídas. É necessário então distribuir estes dados de acordo com a execução das tarefas *Reduce*. Os gerenciadores de tarefa têm por premissa manter o gerenciador de trabalhos informado sobre o estado de execução de suas tarefas. Por outro lado, o gerenciador de trabalhos sabe quais partições das saídas das tarefas *Map* são necessárias para a execução de uma determinada tarefa *Reduce*. Dessa forma, o gerenciador de trabalhos sabe quando uma tarefa *Reduce* atribuída a um nó pode começar. À medida que os dados de entrada uma tarefa *Reduce* vão sendo disponibilizados pelos gerenciadores de tarefa, uma cópia destes dados é feita para o nó que irá executar a tarefa. Quando todos os dados de entrada de uma tarefa *Reduce* estão disponíveis, é realizado um processo de ordenação e junção das diversas partes que compõem os dados de entrada e a tarefa inicia sua execução. Note que, desta forma o arcabouço consegue melhorar seu desempenho, pois

ao mesmo tempo em que ainda existem tarefas *Map* sendo executadas, algumas tarefas *Reduce* que já começam a ser executadas.

Finalmente, após a execução de todas as fases: *Map*, *Shuffle* e *Reduce*, os dados finais são escritos diretamente no sistema de arquivos e ficam disponíveis para o usuário que executou o trabalho.

3.6. Estado atual e futuro da pesquisa em Hadoop

Desde seu início, mas especialmente após se tornar um projeto principal da Apache Software Foundation, o Hadoop tem recebido inúmeras contribuições em seu desenvolvimento. Embora o projeto seja apoiado por grandes empresas que utilizam o arcabouço, grande parte das contribuições também é proveniente da comunidade acadêmica. Apesar de o projeto Apache Hadoop possuir um grande conjunto de subprojetos, a maioria das contribuições acadêmicas se concentra no MapReduce e no sistema de arquivos HDFS.

Com relação ao HDFS, diversas melhorias no sistema de arquivos têm sido propostas por meio da criação de variantes do original. O GreenHDFS (Kaushik, Bhandarkar, & Nahrstedt, 2010) leva em consideração a adaptabilidade e a capacidade de conservação de energia em um aglomerado rodando o Hadoop. O QDFS (Guang-Hua, Jun-Na, Bo-Wei, & Yao, 2011) é uma variante que aplica políticas de cópias de segurança baseadas em uma estratégia de distribuição de dados direcionada à qualidade. DiskReduce (Fan, Tantisiriroj, Xiao, & Gibson, 2009) é uma modificação do HDFS que habilita a compressão assíncrona dos dados replicados para o uso de RAID. O Gfarm (Mikami, Ohta, & Tatebe, 2011) é uma modificação do HDFS que o torna compatível com as normas POSIX, facilitando o acesso direto aos dados. Finalmente, o EDFS (Fesehaye, Malik, & Nahrstedt, 2009) propõe um sistema de arquivos distribuídos semi-centralizado com esquema de balanceamento de carga e compartilhamento de recursos com modificações feitas ao NameNode do HDFS.

O crescimento do paradigma MapReduce e a popularização do arcabouço Hadoop também podem ser comprovados pelo crescente número de trabalhos acadêmicos publicados nas principais bibliotecas digitais como ACM e IEEE. Uma busca no Google Scholar, por exemplo, irá devolver mais de 5000 entradas para Hadoop MapReduce e aproximadamente 2500 entradas para Hadoop HDFS. Desse grande grupo de resultados, algumas propostas visam melhorar o desempenho do arcabouço. Uma estratégia proposta (Abad, Lu, & Campbell, 2011) é posicionar os dados o mais próximo possível do local de processamento, evitando assim a sobrecarga na comunicação dos nós do aglomerado. Nesse sentido, alguns trabalhos têm surgido com o intuito de melhorar os escalonadores de tarefas para se aproveitar melhor da localidade de dados (Zhang, Feng, Feng, Fan, & Ming, 2011; Jin, Luo, Song, Dong, & Xiong, 2011).

O Hadoop adota por padrão a linguagem Java, disponibilizando diversas APIs que podem ser utilizadas para a criação de programas MapReduce. Entretanto, é crescente o interesse por disponibilizar o acesso ao arcabouço por outras linguagens de programação. Além do Hadoop *Streaming*, que permite usar qualquer arquivo executável ou de *script* como classe *mapper* ou *reducer* em uma execução MapReduce, outras propostas ainda sugerem substituir a linguagem Java. O Pydoop (Leo & Zanetti, 2010) propõe uma API para o MapReduce e o HDFS usando a linguagem Python.

O Hadoop também tem sido explorado em áreas mais recentes como computação em nuvem e composição e gerenciamento de serviços Web. Algumas abordagens tratam da seleção distribuída de serviços Web de grande escala baseada em quesitos de qualidade (Pan, Chen, Hui, & Wu, 2011). Outro trabalho propõe um arcabouço de gerenciamento de serviços Web baseado no Hadoop (Zhu & Wang, 2011). Essa proposta de arcabouço utiliza o MapReduce, o HDFS e o HBase em camadas para implementar o gerenciamento e composição de serviços Web. Por fim, Pepper (Krishnan & Counio, 2010) é um sistema com o intuito de prover isolamento de aplicações durante sua execução e escalabilidade dinâmica em máquina de um aglomerado usando Hadoop e ZooKeeper na nuvem.

O Hadoop tem sido um tópico amplamente explorado, especialmente pela sua flexibilidade, podendo ser utilizado como infraestrutura nas mais diversas áreas da ciência, e também pela facilidade de uso e robustez proporcionada pelo projeto. Embora adotado por grandes empresas, o Hadoop pode sofrer em termos de desempenho. Nesse sentido, existem diversos trabalhos estudando estratégias para se obter melhor desempenho na execução de aplicações MapReduce no Hadoop (Joshi, 2012; Kim, Won, Han, Eom, & Yeom, 2011; Ding, Zheng, Lu, Li, Guo, & Guo, 2011; Wlodarczyk, Han, & Rong, 2011; Jiang, Ooi, Shi, & Wu, 2010; Lee, Lee, Choi, Chung, & Moon, 2012). Mesmo com um grande aumento no número de publicações envolvendo o Hadoop, ainda existem tópicos inexplorados e que serão alvo de estudos futuros, como uma melhor integração do Hadoop com outras plataformas e a disponibilização de seus serviços na nuvem.

Referências

Abad, C., Lu, Y., & Campbell, R. (sept. de 2011). DARE: Adaptive Data Replication for Efficient Cluster Scheduling. *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, (pp. 159-168).

Across, P., & Hardware, H. (2008). HDFS Architecture. *Access*, 1-14.

Apache Hadoop. (s.d.). Acesso em 12 de 02 de 2012, disponível em http://hadoop.apache.org

Apache Mahout. (s.d.). Acesso em 24 de 04 de 2012, disponível em http://mahout.apache.org

Big Data University. (s.d.). Acesso em 20 de 02 de 2012, disponível em http://www.db2university.com

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 13.

Ding, M., Zheng, L., Lu, Y., Li, L., Guo, S., & Guo, M. (2011). More convenient more overhead: the performance evaluation of Hadoop streaming. *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (pp. 307-313). New York, NY, USA: ACM.

Fan, B., Tantisiriroj, W., Xiao, L., & Gibson, G. (2009). DiskReduce: RAID for data-intensive scalable computing. *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (pp. 6-10). New York, NY, USA: ACM.

Fesehaye, D., Malik, R., & Nahrstedt, K. (2009). EDFS: a semi-centralized efficient distributed file system. *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (pp. 28:1--28:2). New York, NY, USA: Springer-Verlag New York, Inc.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. (C. Roisin, E. V. Munson, & C. Vanoirbeek, Eds.) *ACM SIGOPS Operating Systems Review, 37*(5), 29.

Guang-Hua, S., Jun-Na, C., Bo-Wei, Y., & Yao, Z. (june de 2011). QDFS: A quality-aware distributed file storage service based on HDFS. *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, *2*, pp. 203-207.

Ibm. (2011). Bringing big data to the Enterprise. *IBMcom*.

Jiang, D., Ooi, B. C., Shi, L., & Wu, S. (#sep# de 2010). The performance of MapReduce: an in-depth study. *Proc. VLDB Endow., 3*(1-2), 472-483.

Jin, J., Luo, J., Song, A., Dong, F., & Xiong, R. (may de 2011). BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, (pp. 295-304).

Joshi, S. B. (2012). Apache hadoop performance-tuning methodologies and best practices. *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering* (pp. 241-242). New York, NY, USA: ACM.

Kaushik, R. T., Bhandarkar, M., & Nahrstedt, K. (2010). Evaluation and Analysis of GreenHDFS: A Self-Adaptive, Energy-Conserving Variant of the Hadoop Distributed File System. *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science* (pp. 274-287). Washington, DC, USA: IEEE Computer Society.

Kim, S., Won, J., Han, H., Eom, H., & Yeom, H. Y. (#dec# de 2011). Improving Hadoop performance in intercloud environments. *SIGMETRICS Perform. Eval. Rev., 39*(3), 107-109.

Krishnan, S., & Counio, J. C. (2010). Pepper: An Elastic Web Server Farm for Cloud Based on Hadoop. *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science* (pp. 741-747). Washington, DC, USA: IEEE Computer Society.

Lam, C. (12 de 2010). *Hadoop in Action.* Manning Publications.

Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., & Moon, B. (#jan# de 2012). Parallel data processing with MapReduce: a survey. *SIGMOD Rec., 40*(4), 11-20.

Lee, K.-R., Fu, M.-H., & Kuo, Y.-H. (june de 2011). A hierarchical scheduling strategy for the composition services architecture based on cloud computing. *Next Generation Information Technology (ICNIT), 2011 The 2nd International Conference on*, (pp. 163-169).

Leo, S., & Zanetti, G. (2010). Pydoop: a Python MapReduce and HDFS API for Hadoop. *Proceedings of the 19th ACM International Symposium on High*

Performance Distributed Computing (pp. 819-825). New York, NY, USA: ACM.

Lin, J., & Dyer, C. (2010). *Data-Intensive Text Processing with MapReduce.* Morgan & Claypool Publishers.

Malley, O. O. (2008). TeraByte Sort on Apache Hadoop. *Whitepaper*(May), 1-3.

Mikami, S., Ohta, K., & Tatebe, O. (2011). Using the Gfarm File System as a POSIX Compatible Storage Platform for Hadoop MapReduce Applications. *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing* (pp. 181-189). Washington, DC, USA: IEEE Computer Society.

National Climate Data Center. (s.d.). Acesso em 15 de 05 de 2012, disponível em http://www.ncdc.noaa.gov/oa/ncdc.html

Pan, L., Chen, L., Hui, J., & Wu, J. (july de 2011). QoS-Based Distributed Service Selection in Large-Scale Web Services. *Services Computing (SCC), 2011 IEEE International Conference on*, (pp. 725-726).

Rogers, Shaw. (2011). Big data is Scaling BI and Analytics. Brookfield.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., et al. (2010). Hive – A Petabyte Scale Data Warehouse Using Hadoop. (F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, et al., Eds.) *Architecture*, 996-1005.

Venner, J. (2009). *Pro Hadoop.* Berkely, CA, USA: Apress.

White, T. (2010). *Hadoop: The Definitive Guide.* O'Reilly Media.

Wlodarczyk, T. W., Han, Y., & Rong, C. (2011). Performance Analysis of Hadoop for Query Processing. *Proceedings of the 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications* (pp. 507-513). Washington, DC, USA: IEEE Computer Society.

Yahoo! Developer Network. (s.d.). Acesso em 23 de 02 de 2012, disponível em http://developer.yahoo.com/blogs/hadoop/

Zhang, X., Feng, Y., Feng, S., Fan, J., & Ming, Z. (dec. de 2011). An effective data locality aware task scheduling method for MapReduce framework in heterogeneous environments. *Cloud and Service Computing (CSC), 2011 International Conference on*, (pp. 235-242).

Zhu, X., & Wang, B. (june de 2011). Web service management based on Hadoop. *Service Systems and Service Management (ICSSSM), 2011 8th International Conference on*, (pp. 1-6).