INE5645 – PROGRAMAÇÃO PARALELA E DISTRIBUÍDA

UNIDADE V

UM EXEMPLO DIDÁTICO DE APLICAÇÃO HADOOP

3.1 INTRODUÇÃO

O grande avanço na última década, do 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 framework (arcabouço) para o armazenamento e processamento paraleleo e distribuído 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. A Unidade V apresentará os conceitos principais do **Apache Hadoop**, demonstrando também suas características, vantagens, aplicações e componentes dessa ferramenta.

3.2 Framework Apache Hadoop

Hadoop é um *framework* (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.

Formas de execução do Hadoop

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.

A tarefa prática proposta para a Unidade V, deve ser inicialmente feita, usando o modo local, em que uma única máquina comportará a instalação do Hadoop e a execução de uma aplicação didática explicada na última etapa deste material.

Para se entender os modos de execução seguintes, precisamos visualizar a Figura 3.3 que mostra os processos componentes do Framework Hadoop, que serão explicados mais adiante na seção

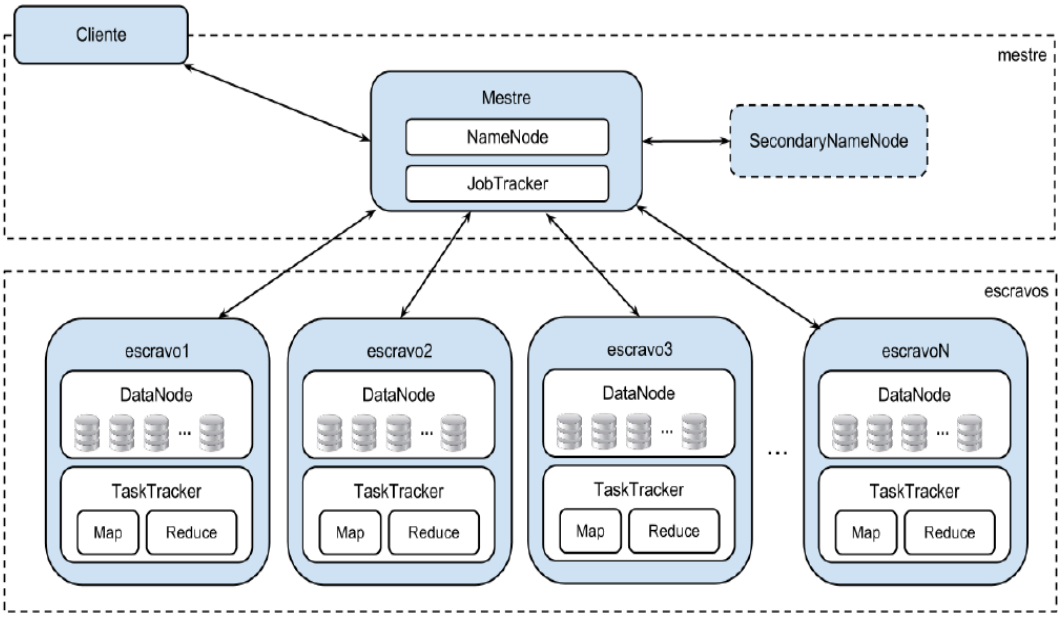


Figura 3.3. Processos do Hadoop

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.

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

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>
```

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 – O SISTEMA DE ARQUIVOS DISTRIBUÍDO DO HADOOP

O **Hadoop Distributed File System** (HDFS) é um sistema de arquivos distribuído integrado ao arcabouço **Hadoop**. Esse sistema teve forte inspiração no *Google File System*, o sistema de arquivos distribuído 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 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.

Os Comandos do 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.

O Processamento

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, **`meuarquivo.txt,`** deve ser passado o nome do arquivo que deseja copiar do HDFS, com o seu respectivo caminho. O segundo parâmetro, **`localfile,`** 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 *framework* (arcabouço Hadoop).

3.3.2 COMPONENTES DO HADOOP – OS PROCESSOS DO HADOOP

Uma execução típica de uma aplicação Hadoop em um aglomerado (cluster) utiliza cinco processos diferentes: NameNode, DataNode, SecondaryNameNode, JobTracker e TaskTracker.

Os três primeiros (NameNode, DataNode, SecondaryNameNode) são integrantes do modelo de programação *MapReduce*, e os dois últimos (JobTracker e TaskTracker) 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 de um cluster, onde rodam os escravos.

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 (mestre) contém o NameNode, o JobTracker e possivelmente o SecondaryNameNode. Já o segundo (escravos), 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.

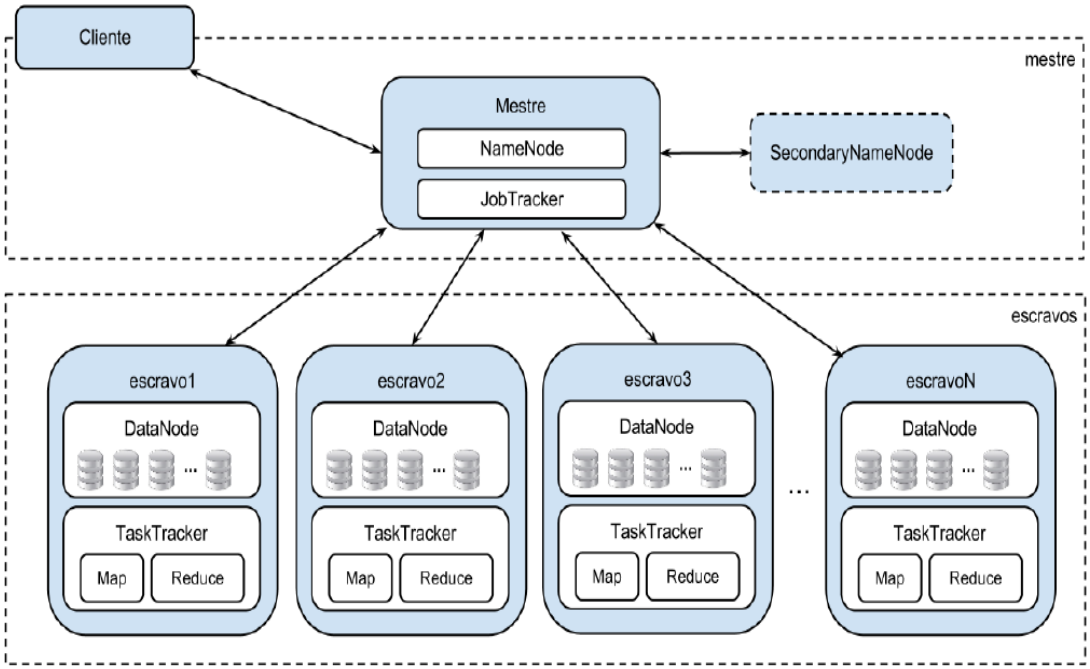


Figura 3.3. Processos do Hadoop

- 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.

3.3.3 ARQUITETURA HDFS

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 cluster (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 (*rack*) é utilizada como um conjunto de máquinas alocadas em um mesmo espaço físico e interligadas por um comutador (*switch*).

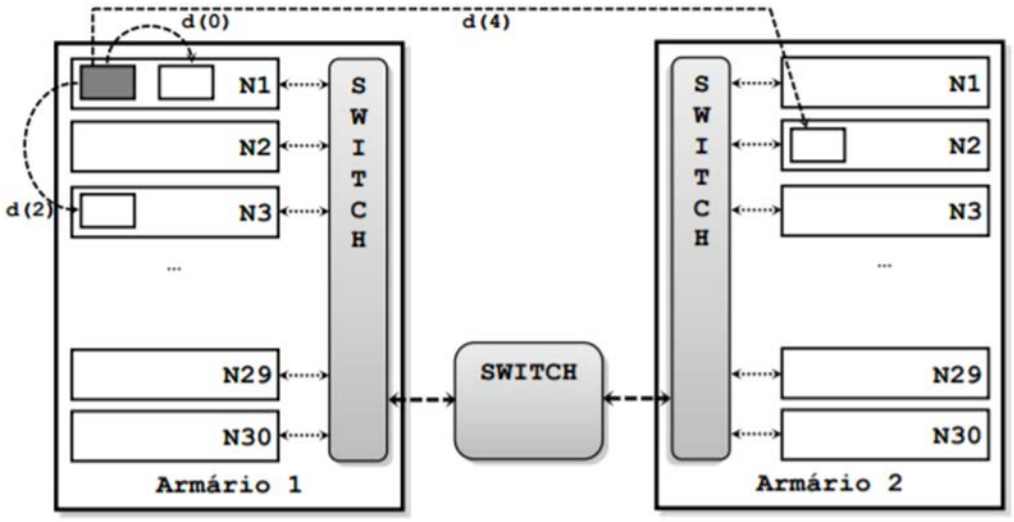


Figura 3.12. Topologia de rede de um típico aglomerado executando Hadoop

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, o *FsImage*, 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, e 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 ele 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.

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 (cluster) de máquinas.

Embora funcional, essa tarefa seria muito difícil de ser implementada sem a utilização de um recurso específico, como o HDFS.

Com o HDFS, 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 (Conteúdo do arquivo *masters*) e 3.10 (Conteúdo do arquivo slaves).

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.

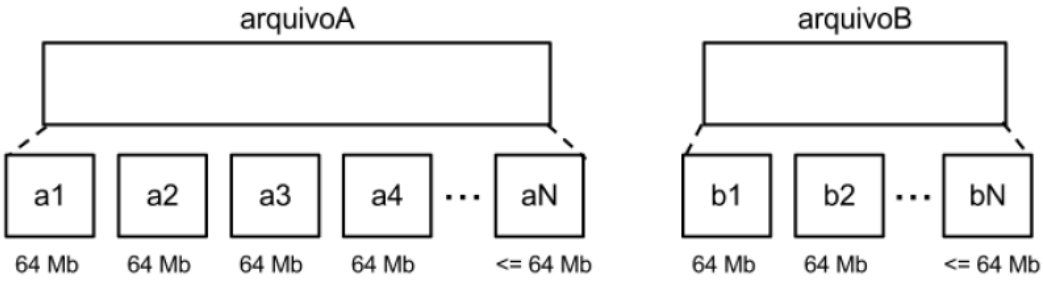
Uma outra característica interessante do HDFS é que, no caso de um dado não ocupar todo o tamanho do bloco reservado a ele, o espaço restante não é perdido, podendo ser utilizado por outros dados.

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 o *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 armários 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.

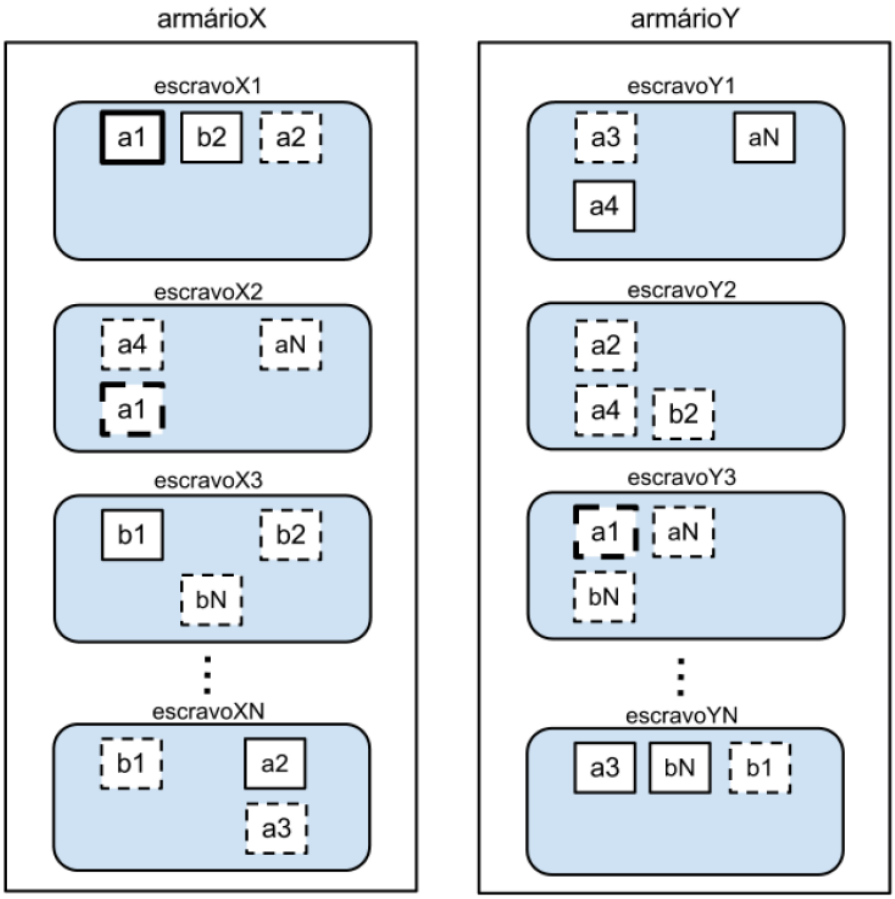


Figura 3.4. Replicação de blocos de dados

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 (LIST PROCESSING).

Uma lista é a estrutura de dados fundamental desta linguagem. LISP é uma família de linguagens de programação – existem alguns dialetos da linguagem. Foi concebida por John McCarthy em 1958. Num célebre artigo, ele mostra que é possível usar exclusivamente **funções matemáticas** como **estruturas de dados elementares** (o que é possível a partir do momento em que há um mecanismo formal para manipular funções. Durante os anos de 1970 e 1980, LISP se tornou a principal linguagem da comunidade de inteligência artificial, linguagem interpretada e linguagem funcional. Tanto os dados, como o programa são representados como listas, o que permite que a linguagem manipule o código fonte como qualquer outro tipo de dados.

Para o nosso estudo básico, essas funções (*Map* e *Reduce*) 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}, (x 2)) **>** {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** (x 2) 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 distribuída.

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 *Cluster* (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.

Grande parte do sucesso dessa recriação do *MapReduce* foi alavancada pelo desenvolvimento do sistema de arquivos distribuído Google File System (GFS), cujos conceitos foram aproveitados para gerar as versões preliminares do arcabouço Apache Hadoop.

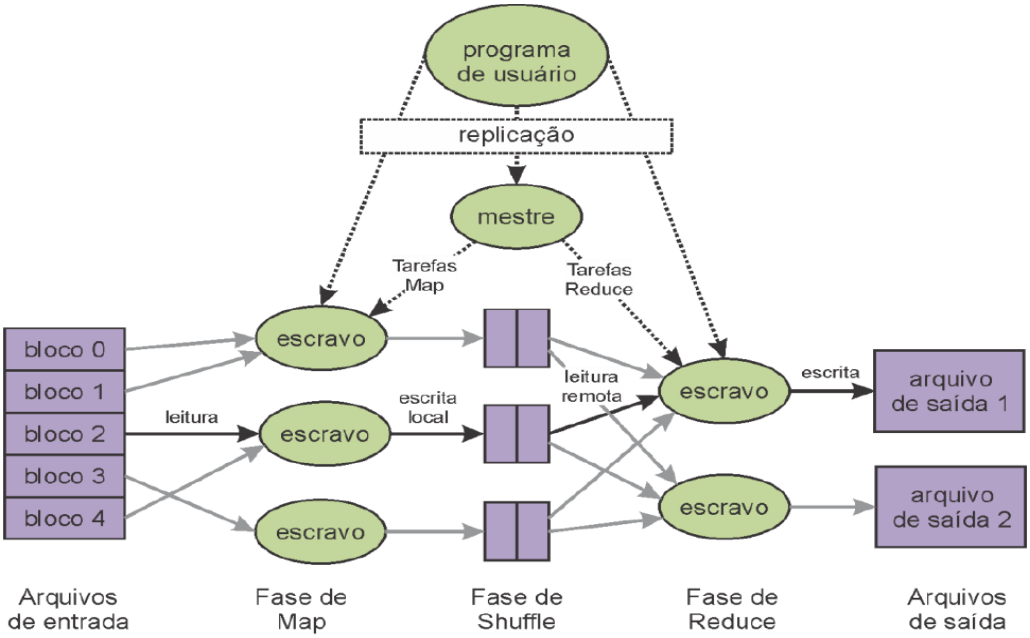


Figura 3.5. Modelo MapReduce implementado pela Google (Adaptado de: Dean & Ghemawat, 2008)

O Hadoop *MapReduce* pode ser visto como um paradigma de programação que expressa computação distribuída como uma sequência de operações distribuídas em um 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)

3.5 EXEMPLO DIDÁTICO DE APLICAÇÃO

A descrição do paradigma de programação Hadoop 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 dados de 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 de entrada "entrada1.txt" e "entrada2.txt".

O *framework* (arcabouço) é chamado e como parâmetro recebe um arquivo JAR (**J**ava **AR**chive).

Dentro do arquivo .jar, 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 *cluster* (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 (*Map*) e a fase de redução (*Reduce*).

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 (*Shuffle*) 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*).

No Quadro 3.17 podemos ver como a classe TokenizerMapper estende a classe Mapper (Mapper). 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).

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

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, onde também são especificados pares de chaves/valores tanto para entrada como para saída.

Na implementação do exemplo podemos verificar que o método IntSumReducer estende a classe Reducer (Reducer<Text, IntWritable, Text, IntWritable>). Isso 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 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 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.

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

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 podem ser chaves ou valores dentro do *framework* Hadoop (arcabouço).

Essa última é uma combinação das interfaces Writable e java.lang.Comparable. 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).

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

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.

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 *Framework* Hadoop trabalha com o conceito de Job (trabalho), 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 **coredefault.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 **Configuration**, 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 cls)**, a classe **Combiner** pelo método **setCombinerClass(Class cls)** e a classe **Reducer** pelo método **setReducerClass(Class 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 o tipo 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 *cluster* (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.

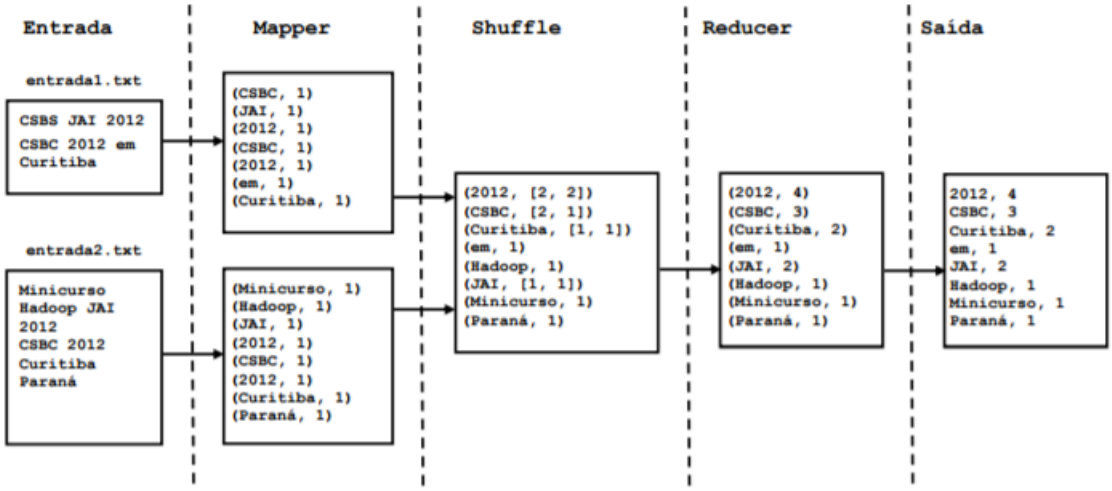


Figura 3.6. Fluxo lógico de execução da aplicação MapReduce do exemplo WordCount

O fluxo lógico de execução dessa aplicação pode ser facilmente visualizado na Figura 3.6, devido ao seu baixo grau de complexidade. As fases podem ser bem definidas e divididas. Isso ocorre também porque o *framework* Hadoop (arcabouço) esconde (para tornar a execução da aplicação o mais transparente possível) 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 a troca de mensagens de maneira a executar o trabalho (*Map, Shuffle, Reduce*).

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.

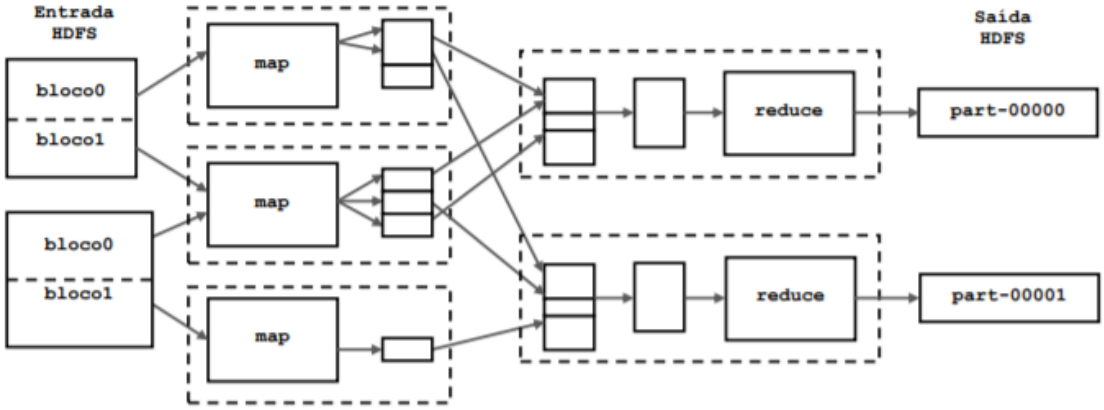


Figura 3.7. Fluxo de execução física de uma aplicação MapReduce no Hadoop

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

==========

Apache Hadoop – É hoje que vai instalar o seu primeiro cluster?

https://pplware.sapo.pt/linux/apache-hadoop-hoje-vai-instalar-primeiro-cluster/

Itens também pesquisados

Como instalar o hadoop no windows

Instalar hadoop no ubuntu

Hadoop download

Como baixar o hadoop

Ambiente hadoop

Apache hadoop

Como funciona o hadoop

Hadoop tutorial portugues

Hadoop: fundamentos e instalação - DevMedia

Instalando Apache Hadoop [Artigo] - Viva o Linux

Hadoop Tutorial - Installing a Hadoop Cluster - YouTube

INTRODUÇÃO AO HADOOP + INSTALANDO HADOOP DE FORMA DISTRIBUÍDA

https://mariannelinharesbr.wordpress.com/2016/06/14/introducao-ao-hadoop-instalando-hadoop-de-forma-distribuida/

How to Install HADOOP - 100% WORKING - YouTube

Instalando o Hadoop – Marcel Ferry

INSTALAÇÃO E CONFIGURAÇÃO DO HADOOP (SINGLE NODE)

http://blog.marcoreis.net/instalacao-e-configuracao-do-hadoop-single-node/

O que é um Cluster ?

Cluster é um termo em inglês que significa “aglomerar” ou “aglomeração” e pode ser aplicado em vários contextos. No caso da computação, o termo define **uma arquitetura de sistema capaz combinar vários computadores para trabalharem em conjunto** ou pode denominar **o grupo em si de computadores combinados**.

Cada estação é denominada “**nodo**” e, combinadas, formam o **cluster**. Em alguns casos, é possível ver referências como “supercomputadores” ou “computação em cluster” para o mesmo cenário, representando o hardware usado ou o software especialmente desenvolvido para conseguir combinar esses equipamentos. Não é necessário haver um conjunto de hardware exatamente igual em cada nó. Por outro lado, é importante que todas as máquinas utilizem o mesmo sistema operacional, de forma a garantir que o software que controla o cluster consiga gerenciar todos os computadores que o integram.

Processamento paralelo e distribuído

Uma das principais categorias de cluster é o tipo de aglomerado, em que grandes tarefas são divididas em atividades menos complexas, distribuídas pelo sistema como um todo e, executadas paralelamente pelos vários nodos do cluster. Então, a aplicabilidade mais eficiente desse tipo é em caso de tarefas computacionais muito complexas em que **Hadoop** pode ser utilizado.

Você pode esclarecer outros conceitos básicos sobre clusters, lendo o link:

https://www.opservices.com.br/o-que-e-um-cluster/

https://www.infowester.com/cluster.php

https://pt.wikipedia.org/wiki/Cluster

Criação de um cluster - Amazon Elastic Container Service

Manual de montagem de um Cluster Beowulf...

Anatomia de um programa Hadoop

trabalho

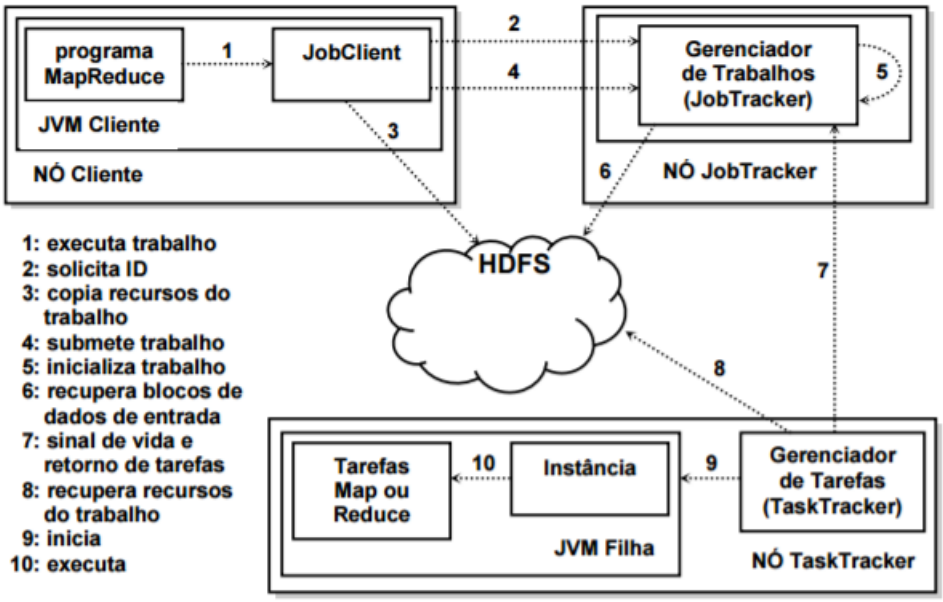


Figura 3.11. Modelo de execução de um trabalho MapReduce no Hadoop (Adaptado de: White, 2010)

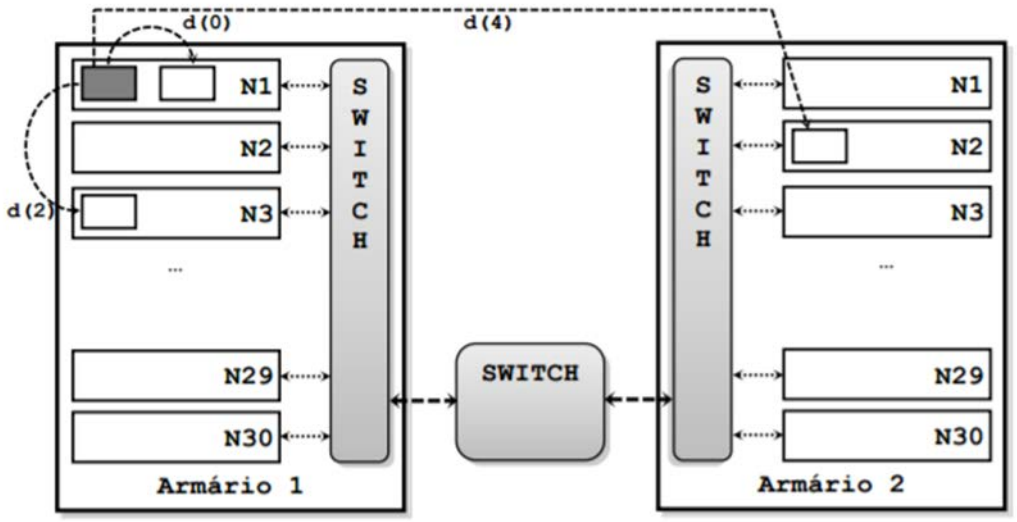


Figura 3.12. Topologia de rede de um típico aglomerado executando Hadoop

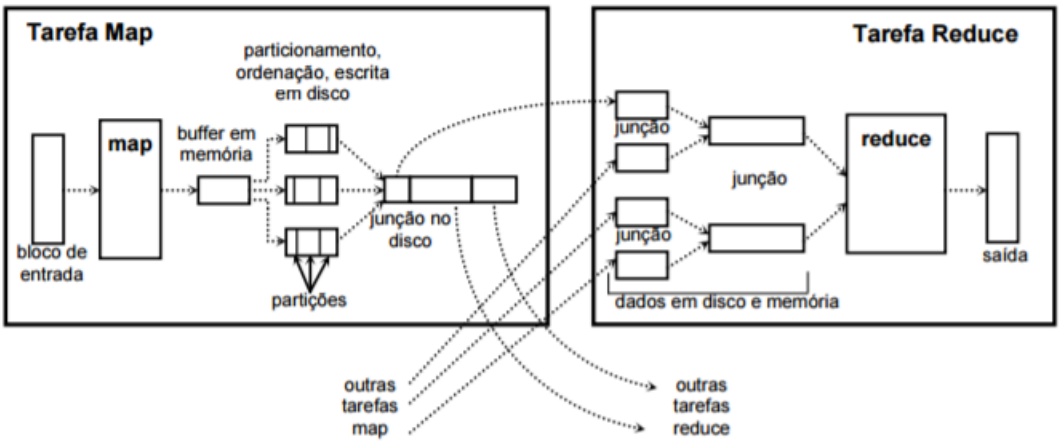


Figura 3.13. Ilustração das fases de *Shuffle* e *Sort* no Hadoop (Adaptado de: White, 2010)

SISTEMA DE ARQUIVOS DISTRIBUIDO

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) da empresa Red Hat;
- Google File System (GFS)
- . . .

O GFS, desenvolvido pela Google em 2003, 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.