FUNDAMENTOS DE BIG DATA

Thiago Nascimento Rodrigues

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

MapReduce na prática: desenvolvimento e testes locais em Java

OBJETIVOS DE APRENDIZAGEM

- Descrever o processo de implementação de uma solução MapReduce em Java utilizando Maven e IDE.
- Justificar, por meio de casos de uso, o processo de desenvolvimento de soluções MapReduce em Java.
- Reconhecer o processo de testes de aplicação MapReduce de maneira local.

Introdução

O MapReduce é um paradigma largamente utilizado no contexto de processamento de *big data*. Uma das implementações mais utilizadas desse modelo é o Apache Hadoop, que oferece grande flexibilidade para que aplicações consigam explorar os seus recursos. Nesse sentido, a compreensão dos detalhes essenciais relacionados com a construção de aplicações sobre o Hadoop se torna fundamental para que cenários mais complexos possam ser corretamente modelados e implementados nessa infraestrutura.

Neste capítulo, você vai estudar o processo de implementação de uma aplicação MapReduce utilizando a linguagem de programação Java. Alguns recursos mais avançados, como o uso do gerenciador de dependências Maven e do ambiente de desenvolvimento integrado (IDE, do inglês *integrated development environment*) NetBeans, farão parte de todo o código apresentado. Além disso, você vai verificar

em detalhes a construção de uma aplicação real, desde as configurações iniciais do ambiente até a execução da solução concluída. Você também vai compreender como ocorre a validação de toda a lógica de código implementada.

Aplicações MapReduce em Java

MapReduce é o modelo de processamento primário suportado pela versão inicial do Apache Hadoop. O processamento de dados executado pelo MapReduce segue um modelo de divisão e conquista e é fundamentado nos conceitos de programação funcional, além de pesquisas em banco de dados. O próprio nome MapReduce faz referência às duas etapas distintas que são aplicadas a todos os dados de entrada, a saber: uma função de **mapeamento** (do inglês *map*) e uma função de **redução** (do inglês *reduce*) (DEAN; GHEMAWAT, 2004).

Toda aplicação MapReduce é composta por uma sequência de tarefas construídas sobre esse modelo. Em cenários mais complexos, é possível que exista uma aplicação principal que exige a execução de várias tarefas, em que a saída do estágio de redução de uma tarefa constitui a entrada para a etapa de mapeamento de outra tarefa subsequente. Além disso, pode haver múltiplas funções de mapeamento ou redução. Apesar da variedade de configurações que podem ser construídas, os conceitos centrais do modelo permanecem os mesmos. A Figura 1 apresenta uma visão esquemática de como as funções `map` e `reduce` são empregadas no modelo MapReduce.

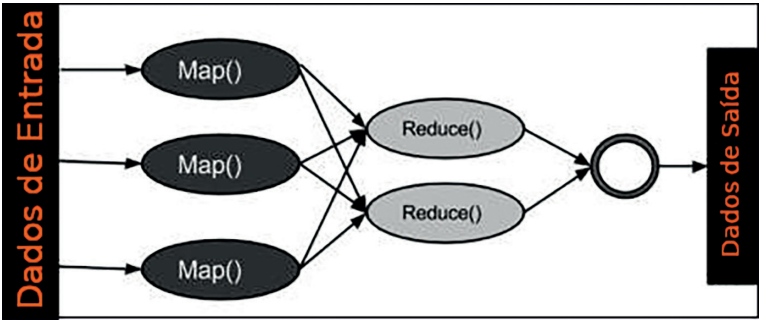

Figura 1. Fases `map` e `reduce`.
Fonte: Adaptada de Tutorials Point (2020).

Em uma aplicação construída segundo o modelo MapReduce, os dados de origem (ou de entrada) são divididos em "pedaços" menores ou registros. Cada pedaço é utilizado para alimentar uma instância específica da função de `map`. Após a aplicação da função em seu respectivo registro, um conjunto de dados intermediário é produzido. Esses registros são, por sua vez, recuperados, e todos os que estão associados entre si por alguma regra previamente definida são repassados para uma função de redução. A saída final da função de redução para todos os conjuntos de registros é o resultado geral do trabalho completo (SITTO; PRESSER, 2015).

Sob uma perspectiva funcional, o modelo MapReduce transforma estruturas de dados de uma lista de pares (chave, valor) em outra lista. Durante a fase de `map`, os dados são carregados do Sistema de Arquivos Distribuído do Hadoop (Hadoop Distributed File System), e uma função é aplicada em paralelo a cada entrada (chave, valor), gerando uma nova lista de pares (chave, valor). De maneira sintética, esse processo pode ser expresso pelo seguinte mapeamento:

```
map(chave1, valor1) → lista(chave2, valor2)
```

Na próxima etapa, o *framework* Hadoop coleta todos os pares com a mesma chave de todas as listas e os agrupa, criando um novo agrupamento para cada chave. Em seguida, uma função `reduce` é aplicada em paralelo a cada grupo, o que, por sua vez, produz uma lista de valores conforme expressado pela relação a seguir:

```
reduce(chave2, lista(valor2)) → chave3, lista(valor3)
```

Desenvolver uma aplicação MapReduce na linguagem de programação Java envolve, essencialmente, conhecer a **interface de programação de aplicação** (API, do inglês *application programming interface*) que é exposta pela biblioteca principal do ecossistema Hadoop, da qual o principal pacote é o `org.apache.hadoop.mapreduce`. De maneira suscinta, construir uma aplicação MapReduce vai demandar as tarefas básicas de:

a) estender as classes base `Mapper` e `Reducer` do Hadoop; e
b) sobrescrever os métodos `map`() e `reduce`(), respectivamente, adicionando a lógica desejada.

Ferramentas de apoio ao desenvolvimento de aplicações, conhecidas como **IDEs**, podem facilitar de maneira significativa o trabalho de codificação. Uma tradicional ferramenta que oferece esse tipo de suporte é o **NetBeans**, que pode ser baixado e instalado gratuitamente (NETBEANS, 2020). Todos os exemplos e casos de uso apresentados neste capítulo serão construídos com base na versão 8.2 do NetBeans. Além disso, será utilizada a versão 8 da linguagem Java.

Fique atento

Uma API é um conjunto de definições e protocolos utilizados para construir e integrar *software*. As APIs permitem que um produto ou serviço se comunique com outros produtos e serviços sem precisar saber como eles são implementados (RED HAT, 2020).

Outro elemento que pode otimizar de maneira considerável o esforço de desenvolvimento de aplicações é um **gerenciador de dependências**. Um componente desse tipo poupa o desenvolvedor da necessidade de descobrir e especificar as bibliotecas (dependências) que são utilizadas pelas próprias bibliotecas do sistema em construção. Em projetos Java, um gerenciador de dependências largamente utilizado é o **Apache Maven**, o qual pode ser baixado e instalado gratuitamente (MAVEN, 2020). Neste capítulo, a versão 3.6.0 do Maven foi utilizada.

Criando aplicação MapReduce no NetBeans

Uma vez que o ambiente esteja com as ferramentas instaladas — Java, NetBeans e Maven —, o primeiro passo para a construção de uma aplicação MapReduce é a criação de um novo projeto no NetBeans. Isso pode ser feito por meio do menu Arquivo > Novo Projeto. Para que o novo projeto tenha suas dependências gerenciadas pelo Maven, é preciso selecionar a categoria Maven, seguida do tipo de projeto Aplicação Java. A Figura 2 apresenta a janela de criação de um novo projeto. O passo subsequente é acionar o botão Próximo, informar o nome do projeto a ser criado e acionar o botão Finalizar. A partir daí, o NetBeans fará a instanciação de um novo projeto Java Maven, e sua estrutura será disponibilizada na aba Projetos.

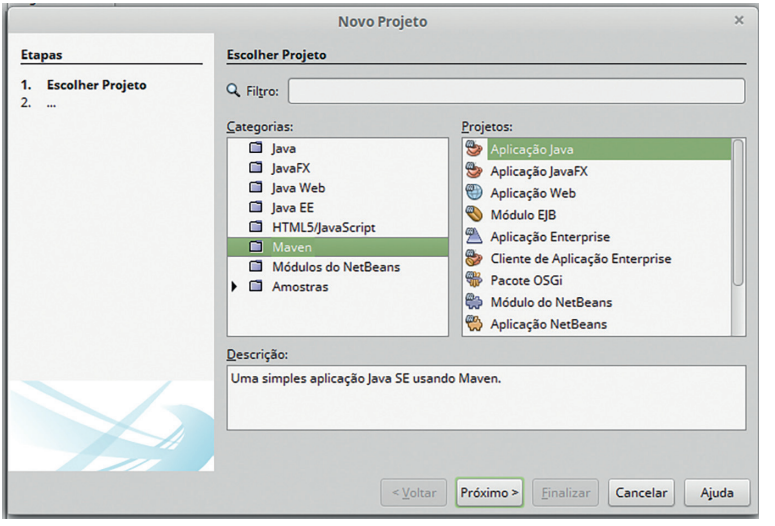

Figura 2. Tela de criação de novo projeto no NetBeans.

Um projeto Java Maven no NetBeans é composto de quatro seções principais, descritas a seguir.

- **Pacotes de código-fonte:** esta seção contém todo o código Java da aplicação.
- **Dependências:** esta seção contém todas as dependências (bibliotecas) utilizadas pela aplicação.
- **Dependências Java:** esta seção contém a referência para o kit de desenvolvimento Java (Java *development kit*) utilizado pelo projeto.
- **Arquivos do projeto:** esta seção contém arquivos auxiliares utilizados pelo projeto. Um dos principais arquivos de um projeto Maven — o `pom.xml` — deve ser localizado nesta seção.

Com a estrutura do projeto montada, é preciso informar as bibliotecas MapReduce que serão utilizadas pelo sistema. Como o foco é no uso da API disponibilizada pelo Hadoop, as dependências necessárias precisam ser informadas no arquivo `pom.xml`. Nesse caso, duas bibliotecas precisam ser indicadas: *hadoop-client* e *hadoop-core*. Até julho de 2020, a versão mais recente da primeira biblioteca era a 3.3.0. Já a segunda biblioteca

não sofre alterações desde 2013. Logo, é recomendável o uso da versão 1.2.1 — a última disponível. A indicação dessas bibliotecas no arquivo *pom.xml* deve ser feita conforme o código apresentado a seguir. As dependências são listadas internas ao elemento raiz do arquivo — `<project/>` — e abaixo dos elementos de propriedades — `<properties/>`. Para que as bibliotecas sejam baixadas a partir dos repositórios remotos, é preciso acionar a construção do projeto pelo Maven por meio do comando `Construir`, que pode ser encontrado invocando o menu de contexto sobre a raiz do projeto.

```
    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>3.3.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-core</artifactId>
            <version>1.2.1</version>
        </dependency>
    </dependencies>
</dependencies>
```

O código Java de uma aplicação MapReducer é centrado na extensão das classes `Mapper` e `Reducer` do Hadoop. Ambas pertencem ao pacote *org.apache.hadoop.mapreduce*. A classe `Mapper` é definida em termos dos tipos de entrada e saída dos pares chave/valor. A função `map` é definida por meio de uma operação de sobrecarga do método `map` da classe. Esse método usa um par chave/valor de entrada como parâmetro.

Outro parâmetro é uma instância da classe `Context` que fornece vários mecanismos de comunicação com o Hadoop, um dos quais é a saída gerada pela aplicação da função (método) `map`. Logo, o corpo desse método deve conter toda a lógica relacionada à operação de mapeamento dos dados de entrada para a lista de pares chave/valor de saída. O código a seguir apresenta como a extensão da classe e a sobrecarga do método podem ser feitos. É importante notar que, inicialmente, os tipos dos dados de entrada

e de saída da classe `Mapper` são genéricos — tipo `Object`. Na prática, esses tipos devem ser alterados, para corresponderem aos tipos dos pares chave/valor de entrada e saída da função `map`.

```
import java.io.IOException;
import org.apache.hadoop.mapreduce.Mapper;

public class PrototipoMapper extends
                    Mapper<Object, Object, Object, Object>{
    @Override
     protected void map(Object key, Object value, Context
context)
              throws IOException, InterruptedException {
         // Lógica a ser adicionada
    }
}
```

É importante observar ainda que, na classe `Mapper`, o método `map` se refere apenas a uma única instância dos pares chave/valor. Esse é um aspecto crítico do paradigma MapReduce, o qual requer que sejam projetadas classes para o processamento de registros únicos. A responsabilidade de transformar um conjunto de dados inteiro em um fluxo de pares chave/valor fica a cargo do *framework* que implementa o modelo MapReduce. Em função disso, não é necessário escrever classes para as etapas de `map` ou `reduce` para lidar com o conjunto de dados completo. Além disso, o Hadoop fornece mecanismos para a leitura e escrita de arquivos nos formatos mais utilizados, o que elimina a necessidade de se escreverem analisadores de arquivo para esses tipos.

A classe base `Reducer` funciona de forma muito semelhante à classe `Mapper` e geralmente requer apenas subclasses para substituir um único método `reduce`. O código a seguir descreve como a extensão da classe pode ser feita. Novamente, é importante observar a definição da classe em termos de um fluxo de dados mais amplo — ela é parametrizada por um tipo de par chave/valor genérico e retorna outro tipo de par chave/valor. Em contrapartida, o método de redução real recebe uma única chave e sua lista de valores associada. O objeto `Context` é novamente o mecanismo para produzir o resultado do método. Os tipos genéricos `Object` tanto da classe como do método devem ser modificados de acordo com o tipo de dado a ser processado e gerado pela aplicação.

```
import java.io.IOException;
import org.apache.hadoop.mapreduce.Reducer;

public class PrototipoReducer extends
                    Reducer<Object, Object, Object, Object>
{
    @Override
   protected void reduce(Object key, Iterable<Object> values,
         Context context) throws IOException, Interrupte-
dException {
        // Lógica a ser adicionada
    }
}
```

O passo final para a criação de uma aplicação MapReduce compreende a construção de uma classe conhecida como `Driver`, responsável pela comunicação com o Hadoop, e a especificação dos elementos de configuração necessários para executar um fluxo de trabalho MapReduce. Isso envolve aspectos como informar ao Hadoop quais classes de `Mapper` e `Reducer` usar, onde encontrar os dados de entrada e em qual formato, e onde colocar os dados de saída e como formatá-los. A lógica da classe `Driver` geralmente existe dentro do método principal da classe escrita para encapsular um fluxo de trabalho MapReduce. O código a seguir apresenta a estrutura de uma classe `Driver`.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import   org.apache.hadoop.mapreduce.lib.output.
FileOutputFormat;

public class AplicacaoMapReducer {

   public static void main(String[] args) throws Exception {

        Configuration conf = new Configuration();
```

```
        Job job = Job.getInstance(conf, "AppMapReducer");
        job.setJarByClass(AplicacaoMapReducer.class);
        job.setMapperClass(PrototipoMapper.class);
        job.setReducerClass(PrototipoReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

No código apresentado, inicialmente é preciso criar uma instância da classe `Configuration`, responsável por armazenar as configurações do fluxo de trabalho MapReduce. Essa instância é passada para o objeto `Job`, que representa o próprio fluxo de trabalho MapReduce. A primeira configuração a ser feita no objeto `Job` é a indicação da classe principal da aplicação — classe que contém o método `main`. Na sequência, as classes `Mapper` e `Reducer` são informadas, assim como as classes que representam os tipos dos pares chave/valor finais da aplicação. Como etapa final de codificação, os caminhos tanto para o diretório que contém os dados de entrada como para o diretório que conterá os dados de saída devem ser informados. A invocação do fluxo de trabalho MapReduce é feita como último comando (WHITE, 2015).

Para que a aplicação possa ser executada, uma opção é empacotar todo o código construído em um arquivo JAR (do inglês Java ARchive, ou arquivo Java). Assumindo que existam arquivos a serem processados no diretório de entrada de dados informados e que o *framework* Hadoop esteja corretamente instalado no ambiente, o comando que executa a aplicação MapReduce tem o seguinte formato:

```
hadoop jar <arquivo JAR> <diretório de entrada> <diretório
de saída>
```

Outra opção é executar o código localmente. Para isso, o diretório onde as classes compiladas foram geradas deve ser atribuído à variável de ambiente `HADOOP_CLASSPATH`. Nesse caso, a execução da aplicação é feita pela chamada à classe principal, isto é, a classe que implementa o método `main`. Considerando que uma aplicação MapReduce tenha uma classe `Driver` de

nome `sagah.AplicacaoMapReducer`, a seguinte sequência de comandos poderia ser usada para executá-la localmente:

```
mvn compile
export HADOOP_CLASSPATH=target/classes
hadoop sagah.AplicacaoMapReducer <diretório de entrada>
<diretório de saída>
```

Tipos de dados do Hadoop

Como aplicações MapReduce lidam com *big data*, objetos estruturados precisam ser serializados em um fluxo de bytes para serem movidos pela rede ou persistidos em disco em um *cluster* e, então, desserializados novamente, conforme necessário. Quando grandes quantidades de dados precisam ser armazenadas e movidas, esses dados precisam ser eficientes e ocupar o mínimo possível de espaço por questões de armazenamento e tempo de movimentação.

As classes nativas do Java não são suficientemente eficientes para esse volume de operações. Logo, versões adaptadas delas fornecem uma abstração otimizada baseada em *arrays* de bytes que representam o mesmo tipo de informação (SITTO; PRESSER, 2015). Essas classes especializadas implementam duas interfaces — `WritableComparable` e `Writable`. A correspondência com as respectivas classes Java é apresentada no Quadro 1.

Quadro 1. Classes empacotadoras (*wrapper*) para os tipos primitivos Java

Tipo primitivo Java	Implementação `Writable`
`boolean`	`BooleanWritable`
`byte`	`ByteWritable`
`short`	`ShortWritable`
`int`	`IntWritable, VintWritable`
`float`	`FloatWritable`
`long`	`LongWritable, VLongWritable`
`double`	`DoubleWritable`

Fonte: Adaptado de White (2015).

Saiba mais

As classes `VIntWritable` e `VLongWritable` do Hadoop são usadas para tipos `integer` e `long`, respectivamente, cujos tamanhos podem variar. Isso possibilita uma otimização do espaço ocupado pelos dados manipulados pelo Hadoop.

Além das classes `Writable` correspondentes aos tipos primitivos Java, o Hadoop dispõe de um conjunto complementar de classes que também implementam as interfaces `WritableComparable` e `Writable` (SIVA , 2014), conforme descrito a seguir.

- `NullWritable`: é um tipo especial que representa um valor nulo. Nenhum byte é lido ou gravado quando um tipo de dados é especificado como `NullWritable`. Portanto, no Mapreduce, uma chave ou um valor pode ser declarado como `NullWritable` quando não precisa usar esse campo.
- `ObjectWritable`: tipo genérico de propósito geral que pode armazenar qualquer objeto como tipos primitivos Java, `String`, `Enum`, `Writable`, nulo ou vetores.
- `Text`: pode ser usado como o equivalente `Writable` de `java.lang.String` e seu tamanho máximo é 2 GB. Ao contrário do tipo de dados `String` do Java, `Text` é mutável no Hadoop.
- `BytesWritable`: é um *wrapper* para um vetor de dados binários.
- `GenericWritable`: é semelhante a `ObjectWritable`, mas suporta apenas alguns tipos. O usuário precisa criar uma subclasse da classe `GenericWritable` e especificar os tipos a serem suportados.

Exemplo de aplicação do MapReduce

Um caso de uso tradicional que possibilita explorar de forma bem detalhada o desenvolvimento de aplicações MapReduce é o problema da contagem de palavras (em inglês, *wordcount*). Esse problema possibilita ter uma visão clara de todo o processo de codificação e configuração já descrito. Além disso, como os resultados podem ser analisados de forma bem direta, o funcionamento do código implementado pode ser facilmente verificado. Uma aplicação MapReduce para esse problema precisa contar o número de ocorrências de cada palavra dentro de um conjunto de arquivos de texto. A Figura 3 apresenta um esquema do fluxo de trabalho de um contador de palavras.

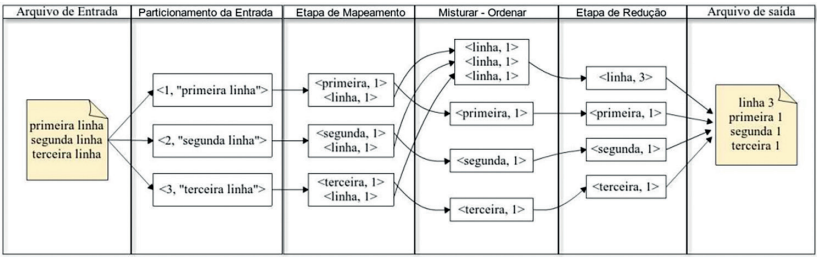

Figura 3. Fluxo de trabalho de um contador de palavras.
Fonte: Adaptada de Bernardes (2015).

É importante observar que, dentre as quatro tarefas encadeadas para o processamento do arquivo de entrada, apenas duas precisam ser explicitamente implementadas, a saber, as etapas de mapeamento e redução. As demais são executados pelo próprio *framework* Hadoop. A saída esperada por uma aplicação MapReduce de contagem de palavras é um conjunto de pares chave/valor, em que a chave corresponde a uma palavra presente no(s) arquivo(s) de entrada e o valor é referente ao número de ocorrências daquela palavra no conjunto de dados informados.

O código a seguir apresenta como a etapa de mapeamento de um contador de palavras pode ser implementada por meio de uma extensão da classe `Mapper` do Hadoop. Cabe ressaltar que a chave da entrada da classe `Mapper` não é efetivamente usada. O `TokenizerMapper` especifica um `TextInputFormat` como o formato dos dados de entrada, e, por padrão, isso entrega ao mapeador os dados nos quais a chave é o deslocamento de byte no arquivo e o valor é o texto dessa linha. Na prática, é muito raro que haja a necessidade de se manipular essa chave de deslocamento de byte, mas ela é fornecida. O mapeador é executado uma vez para cada linha de texto na fonte de dados e sempre que pega a linha e a divide em palavras. Em seguida, ele usa o objeto `Context` para a geração (mais comumente conhecida como emissão) de cada novo par chave/valor no formato (palavra, 1).

```
public class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable> {

    private final IntWritable one = new IntWritable(1);
    private final Text word = new Text();
```

```
    @Override
   public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

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

Saiba mais

`TextInputFormat` é um dos formatos de arquivo do Hadoop. Como o nome sugere, ele é usado para ler linhas de arquivos de texto. Basicamente, ajuda a gerar pares de chave/valor a partir do texto. Inicialmente, os arquivos de texto são divididos em linhas com a ajuda do avanço de linha (movendo uma linha para a frente) ou retorno de carro (movendo o cursor para o início da linha) para verificar o fim da linha — isso é chamado de divisões ou particionamentos. Depois que as divisões são criadas, os pares de chave/valor são gerados com a ajuda do `TextInputFormat`.

Na etapa de redução, o Hadoop executa o `Reducer` uma vez para cada chave, e sua implementação para o contador de palavras simplesmente contabiliza os números no objeto `Iterable` e fornece saída para cada palavra na forma de (palavra, contagem). O código a seguir apresenta o código Java para essa etapa. Analisando as assinaturas do `Mapper` previamente descrito e do `Reducer`, é importante perceber que a classe `TokenizerMapper` aceita `IntWritable` e `Text` como entrada e fornece `Text` e `IntWritable` como saída. Por outro lado, a classe `IntSumReducer` tem `Text` e `IntWritable` aceitos como entrada e saída. Esse é um padrão bastante comum, em que o método `map` executa uma inversão na chave e nos valores e emite uma série de pares de dados nos quais o redutor executa a agregação.

```
public class IntSumReducer extends
       Reducer<Text, IntWritable, Text, IntWritable> {

    private final IntWritable result = new IntWritable();
```

```
    @Override
   public void reduce(Text key, Iterable<IntWritable> values,
                      Context context)
            throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable val: values) {
            sum += val.get();
        }

        result.set(sum);
        context.write(key, result);
    }
}
```

Uma vez implementadas as duas funções `map` e `reduce`, é preciso construir o código da classe `Driver`. O próximo código detalha como o fluxo de trabalho MapReduce do contador de palavras pode ser configurado. Conforme apresentado anteriormente, é preciso indicar o nome da classe principal do arquivo JAR — neste caso, `ContadorPalavras.class`. Na sequência, as classes que implementam as etapas de `map` e `reduce` devem ser indicadas, assim como os tipos dos dados de saída — tanto a chave como o valor. Finalmente, os diretórios onde os dados de entrada se encontram e onde os dados de saída serão armazenados devem ser indicados.

```
public class ContadorPalavras {

   public static void main(String[] args) throws Exception {

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(ContadorPalavras.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
          System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Testando aplicações MapReduce localmente

O caráter funcional das aplicações MapReduce possibilita que as suas etapas sejam adequadamente isoladas para fins de teste e depuração. Para testar uma aplicação localmente, nenhuma configuração adicional precisa ser feita, já que a própria API do Hadoop se encarrega de executar o *cluster* em modo *standalone*.

Assim, para fins de validação do funcionamento do contador de palavras, uma boa estratégia é utilizar dados cuja saída já seja previamente conhecida. Esse é o caso dos arquivos apresentados por Goldman *et al.* (2012), cujos dados são apresentados nas duas primeiras colunas do Quadro 2. Considerando que os arquivos `entrada1.txt` e `entrada2.txt` se encontram no diretório `/input`, que a saída será gerada no diretório `/saida` e que o nome do arquivo JAR gerado pelo Maven é `contador-palavras.jar`, o seguinte comando faz a chamada da aplicação MapReduce desenvolvida:

```
hadoop jar contador-palavras.jar /entrada /saida
```

Como mostrado anteriormente, uma maneira alternativa de se executar uma aplicação Hadoop localmente é pela chamada direta à sua classe `Driver`. No caso do contador de palavras, os comandos a seguir também poderiam ser usados para testar a aplicação:

```
mvn compile
export HADOOP_CLASSPATH=target/classes
hadoop ContadorPalavras /entrada /saída
```

A saída gerada é apresentada na terceira coluna (`part-r-000`) do mesmo quadro. Quando a execução da aplicação MapReduce finaliza com sucesso, um arquivo adicional de nome `_SUCCESS` é gerado no mesmo diretório onde as saídas são armazenadas.

Quadro 2. Arquivos de entrada para o contador de palavras e as saídas geradas por ele

entrada1.txt	entrada2.txt	part-r-000
CSBC JAI 2012 CSBC 2012 em Curitiba	Minicurso Hadoop JAI 2012 CSBC 2012 Curitiba Paraná	2012, 4 CSBC, 3 Curitiba, 2 em, 1 JAI, 2 Hadoop, 1 Minicurso, 1 Paraná, 1

Fonte: Goldman *et al* (2012).

Conforme já descrito, a etapa `map` do contador de palavras cria um conjunto de pares chave/valor para cada entrada informada. Logo, para uma entrada do tipo "CSBC JAI 2012 CSB C", a saída esperada pelo método `map` da classe `TokenizerMapper` corresponde ao seguinte conjunto de pares chave/valor: `{ (CSBC, 1), (JAI, 1), (2012, 1), (CSBC, 1) }`. Com relação à etapa de redução, a classe `IntSumReducer` construída recebe como parâmetros uma palavra (`Text`) e uma lista contendo o número de ocorrências da referida palavra em cada arquivo de entrada. Com base nos dados fornecidos, um exemplo de situação a ser validada tem o seguinte formato:

- Palavra: "CSBC" → Duas ocorrências no primeiro arquivo e uma ocorrência no segundo.
- Palavra: "2012" → Duas ocorrências no primeiro arquivo e duas ocorrências no segundo.

As listas com as respectivas ocorrências das palavras em cada arquivo são construídas pelos mecanismos de MapReduce do Hadoop (cada posição da lista corresponde ao número de ocorrências da palavra em um arquivo de entrada diferente), e elas são repassadas para a classe responsável pela etapa de redução. Nesse caso, após o processo de redução, são esperados os seguintes pares chave/valor: `{ (CSBC, 3), (2012, 4) }`, o que pode ser atestado por meio de uma análise direta do arquivo gerado.

Referências

BERNARDES, G. L. *Desenvolvimento de software no contexto big data*. 2015. Monografia (Graduação em Engenharia de Software) – Universidade de Brasília, Brasília, DF, 2015. Disponível em https://docplayer.com.br/81387030-Desenvolvimento-de-software-no--contexto-big-data.html. Acesso em: 11 set. 2020.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *In*: SYMPOSIUM ON OPERATING SYSTEM DESIGN AND IMPLEMENTATION, 6., 2004, San Francisco. *Anais* [...]. Berkeley: USENIX, 2004. p. 137-150. Disponível em: https://www.usenix.org/legacy/events/osdi04/tech/full_papers/dean/dean.pdf. Acesso em: 11 set. 2020.

GOLDMAN, A. *et al*. Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades. *In*: JORNADAS DE ATUALIZAÇÕES EM INFORMÁTICA, 31., 2012, Curitiba. *Anais* [...]. Curitiba: [UFPR], 2012. p. 88-136. Disponível em: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/JAI_2012_Cap%203_Apache%20Hadoop.pdf. Acesso: 11 set. 2020.

MAVEN. *Apache Maven Project*. 2020. Disponível em https://maven.apache.org/index.html. Acesso em: 11 set. 2020.

NETBEANS. *NetBeans*: o IDE Java de Código Livre. 2020. Disponível em: https://netbeans.org/index_pt_PT.html. Acesso em: 11 set. 2020.

RED HAT. *What is an API?* 2020. Disponível em: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces. Acesso em: 11 set. 2020.

SITTO, K.; PRESSER, M. *Field Guide to Hadoop*: an introduction to Hadoop, its ecosystem, and aligned technologies. Sebastopol, CA: O'Reilly, 2015.

SIVA. *Hadoop data types*. Hadoop Tutorial, 22 Apr. 2014. Disponível em: http://hadooptutorial.info/hadoop-data-types/. Acesso em: 11 set. 2020.

TUTORIALS POINT. *Hadoop*: MapReduce. 2020. Disponível em: https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm. Acesso em: 11 set. 2020.

WHITE, T. *Hadoop*: the definitive guide. 4th ed. Sebastopol, CA: O'Reilly, 2015.

Leitura recomendada

SOFTWARE TESTING FUNDAMENTALS. *Unit Testing*. 2020. Disponível em: http://softwaretestingfundamentals.com/unit-testing/. Acesso: 11 set. 2020.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS