MapReduce Detalhado

Funcionamento e Recursos

Na edição anterior da MundoJ, foi apresentado um artigo introdutório sobre o Hadoop, explicando de forma básica os seus principais componentes, como instalar e configurar um cluster em modo pseudodistribuído, além de um primeiro exemplo básico de programa MapReduce, demonstrando como desenvolver sobre tal perspectiva, e quais características do Hadoop podem ser aproveitadas ao se utilizar tal técnica. Neste artigo será dado sequência ao estudo de programação MapReduce, com exemplos práticos e dicas de uso de recursos da API do Hadoop para o desenvolvimento de algoritmos de análise de dados com desempenho otimizado.

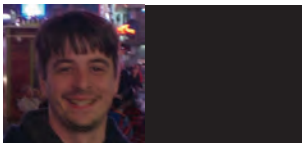
Como mostrado no artigo anterior, um programa MapReduce processa dados na forma de chave/valor. Todo arquivo a ser consumido será dividido em blocos, sendo esses blocos, por sua vez, consumidos paralelamente por tarefas executando em diferentes nós do cluster Hadoop. Também já foi demonstrado que a implementação de um algoritmo de processamento se divide em duas fases, denominadas Map e Reduce, sendo que a lógica de cada uma deve ser implementada em duas classes diferentes, que extendem as classes Mapper e Reducer, respectivamente. Foi explicado ainda que a implementação do Mapper recebe os dados de entrada por meio de seu método map, que é invocado pelo Hadoop, recebendo como parâmetros uma linha do(s) arquivo(s) de entrada, e uma chave de identificação dessa linha (que por padrão é um valor offset da linha do arquivo, mas pode ser configurado para receber um valor do próprio conteúdo do arquivo), e deve gerar, como saída, também uma chave e um valor (através da seguinte chamada: Context.write(key, value)). Entretanto, não foi explicado no artigo anterior que existe uma fase intermediária entre o Map e o Reduce, executada pelo Hadoop, que é o shuffle and sort. Nela, o Hadoop irá agrupar em uma estrutura de coleção todos os valores escritos no Map para um mesmo valor de chave, e depois enviá-los para tarefas da fase Reduce. A Listagem 1 resume como os dados são estruturados no início e fim de cada fase.

Listagem 1. Formato de dados entrada/saída Map/Reduce.

```
map: (Chave1/Valor1) (Chave1/Valor2) -> (ChaveMap1,ValorMap1) (ChaveMap1,ValorMap2)
reduce: (ChaveMap1:[ValorMap1, ValorMap2]) ->(Chave1, ValorFinal1)
```

Wellington Ramos Chevreuil | wellington.chevreuil@gmail.com

Engenheiro de Computação, SCJP, SCJD, SCMAD, SCWCD, SCBCD, SCEA. Atualmente utilizando hadoop e mapreduce em projetos de análise de grande volume de dados e geração de recomendação e demais conteúdos em serviços de entretenimento via web. É também administrador Hadoop certificado pela Cloudera.



Fábio Moreira de Almeida | fabioalmeid@yahoo.com.br

Analista de sistemas formado pela UTAM, especialista em Desenvolvimento em Software Livre pela UEA. Mestrando do curso de Recuperação da Informação pela UFAM. Atualmente, pesquisador do Instituo Nokia de Tecnologia (INdT), trabalhando com o Hadoop em sistemas de recomendação e inteligência contextual. É também administrador Hadoop certificado pela Cloudera.

Flavio Eduardo de Lima | fladulima@gmail.com

Analista de Sistemas, CSM, CSD, SCJP, SCWCD, SCMAD, SCJA. Experiência com hadoop e seu ecossistema em projetos de análise de grande volume de dados. É também administrador Hadoop certificado pela Cloudera.

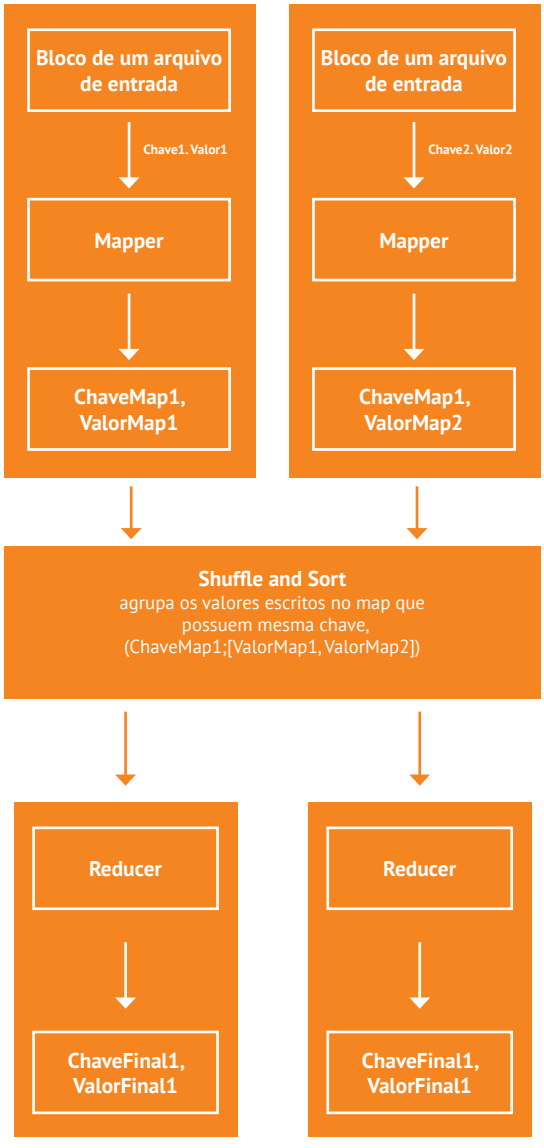


Figura 1. *Fluxo de dados ao longo do MapReduce.*

Na figura 1, é exibido um esquema do fluxo de dados ao longo da execução de um processo MapReduce.

Assim, este artigo abordará em detalhes as diferentes classes da API MapReduce do Hadoop, quando utilizá-las dependendo do tipo de dados a serem processados e de como esses dados estão formatados nos arquivos de entrada. Também explicará as classes auxiliares no controle do fluxo de dados ao longo das fases do processamento, demonstrando com alguns exemplos práticos de aplicação da abordagem MapReduce.

Tipos de Dados

Tanto Mappers quanto Reducers manipulam chaves e valores que são passados como parâmetros pelo Hadoop para as implementações dessas classes, para os métodos map e reduce, respectivamente. Apesar de a API do Hadoop prover certa flexibilidade de se poder tratar as chaves e valores, como números, textos ou mesmo sequências de bytes, a mesma exige que se use tipos próprios, ao invés de classes Java padrão como String ou Wrappers. Isso é necessário porque o Hadoop possui seu próprio esquema de serialização de dados para transferência dos mesmos entre os nós do cluster, sendo esse contrato definido na interface org.apache.hadoop.io.Writable. Existe ainda a interface org.apache.hadoop.io.WritableComparable<T>, que define métodos de comparação entre objetos. WritableComparable deve ser implementada por todo tipo que deva ser usado como chave. Já os valores podem ser de qualquer tipo que implemente Writable.

Assim, há várias implementações de Writable, e praticamente todos os tipos comuns do Java possuem seus equivalentes na API do Hadoop. A figura 2 exibe um diagrama de classes com as implementações de Writable mais usadas.

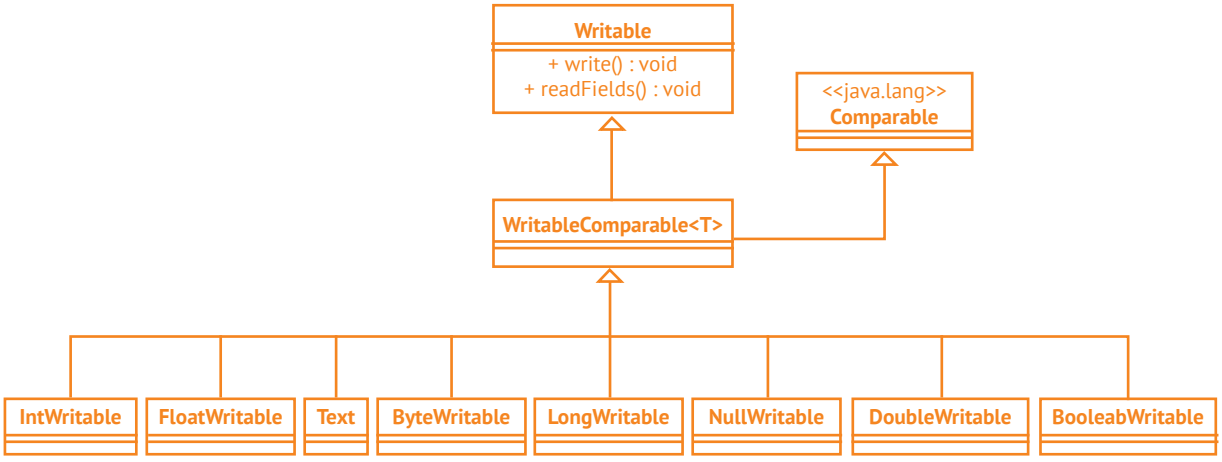


Figura 2. *Principais tipos de dados API Hadoop.*

Conforme pode ser visto no diagrama da figura 2, além de classes similares aos Wrappers do próprio Java, é mostrado também as classes Text e NullWritable. A primeira, como o nome intuitivamente sugere, é útil para manipulação de textos e é similar à classe String. A segunda é apenas um tipo de conveniência a ser usado quando na verdade não há nada a ser passado como parâmetro valor.

Formatos de entrada

O Hadoop pode processar diferentes tipos de formato de dados, desde um simples arquivo-texto até arquivos binários. A forma de representar os dados de um MapReduce específico é definida pela classe InputFormat informada na sua classe Driver. Nesta sessão, serão mostrados algumas das implementações de InputFormat mais utilizadas, mas antes é necessário entender alguns conceitos relacionados a como o Hadoop lê os seus dados de entrada.

Input splits e Registros

Normalmente, os dados estão contidos em arquivos gigantes e sempre que é enviado um novo processamento para o Hadoop, estes arquivos são divididos em pedaços menores para serem consumidos pelo processo de Map. Estes blocos menores são chamados de input splits. Cada input split é dividido em registros. O processo de Map irá ler e processar cada registro do input split. Os input splits são representados pela interface Java InputSplit e um detalhe importante é que a interface contém somente informações sobre a localização e o tamanho do dado, ou seja, o dado de entrada não é armazenado. Para ler os dados do registro para o job, o Hadoop se utiliza da classe RecordReader, onde deve ser definida a lógica de ler os dados do registro e transformá-los nos pares chave/valor a serem passados para a classe Mapper. Para a maioria dos cenários de desenvolvimento de MapReduce, a API do Hadoop já fornece diferentes implementações de InputFormat, InputSplit e RecordReader que podem ser aproveitadas, bastando-se especificar qual o InputFormat desejado. Já em casos mais avançados de uso, é possível customizar completamente a forma de ler e interpretar os dados de entrada implementando essas três classes.

A interface InputFormat

As implementações definidas para essa interface é a maneira com que o Hadoop transforma as linhas dos arquivos de entrada em pares chave/valor.

Entrada com formato de texto

A implementação mais comum para trabalhar com textos é a TextInputFormat. Muito usado quando se quer processar linhas de texto. Na utilização do TextInputFormat cada linha do texto é um registro, onde sua chave é byte offset da linha e o valor é o conteúdo da linha. Para o exemplo de texto da Listagem 2, o Map receberá cada registro em pares de chave/valor, como mostrado na Listagem 3.

KeyValueInputFormat

No exemplo da edição anterior a classe Driver utilizou o KeyValueTextInputFormat (veja linha 41 do código-fonte da classe). Esse é o InputFormat mais usado quando se manipula arquivos-texto.

Listagem 2. Exemplo de arquivo-texto de entrada.

```
Maracatu psicodélico
Capoeira da Pesada
Bumba meu rádio
Berimbau elétrico
Frevo, Samba e Cores
Cores unidas e alegria
Nada de errado em nossa etnia
```

Listagem 3. Exemplo de formatação em chave/valor do texto de entrada.

```
[0, Maracatu psicodélico]
[21, Capoeira da Pesada]
[40, Bumba meu rádio]
[56, Berimbau elétrico]
[64, Frevo, Samba e Cores]
[85, Cores unidas e alegria]
[108, Nada de errado em nossa etnia]
```

Outra implementação da interface InputFormat é a KeyValueTextInputFormat. Nesse caso, para cada registro, torna-se necessário um delimitador que separa a chave do valor. Por padrão, caso não seja definido nenhum delimitador, será usado (\t). Usando KeyValueTextInputFormat cada registro corresponde a uma linha de texto.

Entrada com formato binário

Além do formato texto mostrado anteriormente, o Hadoop pode processar dados em formato binário. A classe SequenceFileInputFormat é uma implementação para processar arquivos com dados binários, todavia é necessário criar os arquivos com os dados usando a classe SequenceFile que provê uma maneira de persistir dados binários usando pares de chave/valor.

Entrada lendo de uma base de dados relacional

A classe DBInputFormat é uma implementação da interface InputFormat e pode ser utilizada para ler dados de uma base de dados relacional. O Job configurado com DBInputFormat usa a API de JDBC para consultar a base de dados relacional. Entretanto, é importante saber que não é recomendado usar essa configuração, pois vários mappers estarão consultando a base simultaneamente ocasionando uma sobrecarga na base. O uso do DBInputFormat só é recomendado em pequenos datasets. Uma alternativa para evitar o uso dessa configuração é importar os dados do banco para o HDFS antes do Job ser processado.

Formatos de saída

Na API do Hadoop, para cada formato de dados de entrada mostrado anteriormente, existe um correspondente para ser usado como saída de dados. A interface que define a saída de dados é OutputFormat. A seguir, serão detalhadas algumas das implementações mais usadas desta interface.

Saída em formato texto

A classe a ser utilizada para que o resultado do processamento de um Job seja em formato texto é TextOuputFormat. O resultado será uma linha de texto escrita no arquivo para cada registro, onde cada um possui uma chave-valor separada por um delimitador que por padrão é o caractere (\t).

Saída com formato binário

Da mesma forma que existe na API do Hadoop uma maneira de ler formato binários, existe também uma maneira de escrever. Nesse caso utiliza-se a classe SequenceFileOutputFormat. As classes da API do Hadoop de manipulação de arquivos binários proporcionam um desempenho melhor do que as que manipulam em formato Texto. Esse formato é bastante utilizado quando o resultado de um processamento serve de entrada para outro processamento.

Saída inserindo em uma base de dados relacional

É muito comum no final do processamento de um Job inserir os resultados em uma base de dados relacional. Para esse caso a API do Hadoop possui a classe DBOutputFormat. Como foi falado anteriormente, o uso da configuração de inserir ou ler de base de dados deve ser evitado para que não haja sobrecarga na base de dados.

Classes Mapper predefinidas

A API do Hadoop provê algumas classes Mapper prontas para uso (ou contendo algum recurso adicional a ser herdado) implementando funcionalidades comuns em alguns cenários de MapReduce. A utilização dessas classes permite que o desenvolvedor se concentre apenas na codificação do Reducer.

Na versão 0.20.2, os seguintes mappers estão disponíveis, todas no pacote org.apache.hadoop.mapreduce.lib:

- **InverseMapper.** Simplesmente inverte os dados de entrada. Tudo que for valor vira chave e tudo que for chave vira valor.
- **MultithreadedMapper.** Esta classe é uma alternativa à classe Mapper padrão, pode ser extendida quando se deseja executar maps em múltiplas threads.
- **TokenCounterMapper.** Clássico cenário de

API antiga

Existe o pacote org.apache.hadoop.mapred contendo vários subpacotes de classes, incluindo mappers e reducers. Todo esse pacote refere-se a versões anteriores do Hadoop, e muito de suas classes e métodos encontram-se deprecadas. Embora o Hadoop 0.20.2 mantenha compatibilidade com esses componentes, o seu uso é extremamente desencorajado.

contagem de palavras. Este mapper efetua um parse do texto do valor, emitindo para cada palavra, um valor inteiro de 1.

Partitioners

Partitioners (particionador) é o processo que determina qual instância de reducer receberá quais pares chave/valor produzidos pelo mapper. Cada mapper precisa endereçar para todos os seus pares chave/valor qual reducer o processará. O leitor pode ser perguntar: "Por que eu não precisei fazer nada disto no exemplo dado na edição anterior da MundoJ?" Isto porque o comportamento padrão do hadoop é aplicar função hash na chave para determinar o seu respectivo reducer. O hadoop força esta estratégia com o uso da classe HashPartitioner. Uma única partição, portanto, refere-se a todos os pares chave/valor que serão enviados para um único reducer. No entanto, em determinadas ocasiões pode ser proveitoso mudar este comportamento padrão bastando apenas customizar o seu partitioner. O partitioner é executado logo após o mapper, porém antes do shuffle and sort e do reducer, como ilustrado na figura 3.

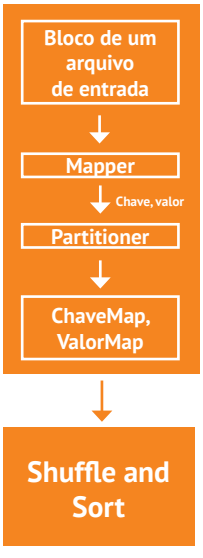


Figura 3. *Ordem do MapReduce – Partitioner.*

Por exemplo, ocasiões nas quais os dados estão bem distribuídos, ou seja, a quantidade de repetições de cada chave é relativamente balanceada em relação às demais, teremos como resultado partições aproximadamente do mesmo tamanho. Entretanto, se a massa de dados está desbalanceada, ou seja, uma chave ocorre muito mais frequentemente que as demais, isto acarretará em um particionamento desbalanceado, fazendo com que alguns reducers tenham mais dados para processar que outros.

Uma possível solução para o problema descrito acima seria escrever um partitioner que dedicasse uma partição única e exclusivamente para a chave que ocorre com maior frequência e distribuir as demais chaves pelas partições compartilhadas restantes. Veja um exemplo de como fazer isto na Listagem 4:

Listagem 4. Classe que customiza um partitioner. Baseado no exemplo dado na edição anterior.

```
public class CensoPartitioner extends
Partitioner<Text,DoubleWritable> {
   @Override
   public int getPartition(Text key, DoubleWritable value,
   int numPartitions) {
      // valor de key = 1980-Sul
      String linha = key.toString();
      String[] valores = linha.split("-");
      return Math.abs(valores[1].hashCode()) %
         numPartitions;
   }
}
```

Este código fará com que os particionamentos sejam criados por região. Ou seja, cada partição conterá uma única região. Poderíamos, por exemplo, criar apenas duas partições, uma para a região que ocorre com maior frequência e outra para todas as demais. Para que o partitioner seja executado é necessário utilizar o método job.setPartitionerClass(CensoPartitioner.class) na classe de configuração comumente denominada Driver (veja edição anterior). Vale ressaltar que um partitioner não muda em nada o resultado de um job, adicioná-lo ou removê-lo não interferirá no resultado final, apenas, distribuirá mais eficientemente (caso bem utilizado) os dados entre os reducers.

Combiners

Combiners (combinadores) permitem que os dados sejam combinados ainda na fase de mapper,

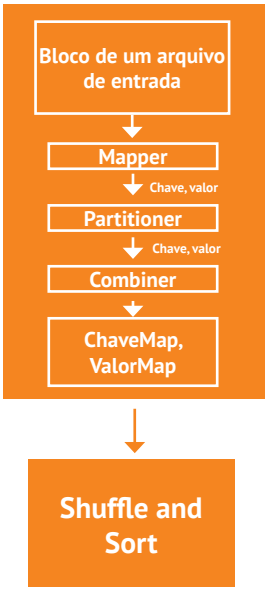


Figura 4. *Ordem do MapReduce – Combiner.*

possibilitando uma redução no output do mapper e consequentemente redução da sobrecarga no tráfego da rede e no processamento do reducer. O combiner é um passo opcional executado logo após o mapper e partitioner, porém antes do shuffle and sort e do reducer, como ilustrado na figura 4.

É possível, por exemplo, através do uso de combiners, ao invés de contar cada ocorrência de uma palavra, já somar todas as ocorrências ainda durante a fase de mapper e passar esta informação calculada para o reducer. Vamos comparar os resultados do processo de mapper para este exemplo sem o uso de combiners e com o uso de combiners na Listagem 5.

Utilizando o exemplo da edição anterior cujo objetivo era calcular a média salarial por região, em cada ano de pesquisa do censo, poderíamos fazer com que o combiner já somasse os salários por região, obviamente sempre levando em consideração o número de elementos somados para que seja possível realizar o cálculo da média. Como cálculo de média não é uma operação distributiva, precisaremos de uma pequena alteração em nosso mapper e reducer, que ficariam como nas Listagens 6 e 7, respectivamente.

Listagem 6. Classe CensoMapper modificada para uso com Combiners. Alteração em vermelho.

```
public class CensoMapper extends
Mapper <Text, Text, Text, Text>{
   @Override
   public void map(Text key, Text value, Context context)
   throws IOException, InterruptedException {
      // chave=>1980, valor=> 31#Sul#Santa
      // Catarina#Xanxere#1700.50
      String linha = value.toString();
      String[] valores = linha.split("#");
      Text chaveSaidaMapper = new Text();
      // monta a chave resultante como [ano]-[região]
      //ex: 1980-Sul
      chaveSaidaMapper.set(key.toString() + "-" +
         valores[1]);

      context.write(new Text(chaveSaidaMapper),
            new Text(valores[4] + ",1"));
   }
}
```

Listagem 7. Classe CensoReducer modificada para uso com Combiners. Alteração em vermelho.

```
public class CensoReducer extends
Reducer <Text, Text, Text, DoubleWritable> {
   public void reduce(Text key, Iterable<Text> values,
   Context context) throws IOException,
   InterruptedException{
```

Listagem 5. Comparação entre saídas de diferentes mappers com e sem uso de combiners.

Sem combiners:

mapper1	("um", 1)	("prato", 1)	("de", 1)	("trigo", 1)	("para", 1)	("um", 1)	("tigre", 1)
mapper2	("dois", 1)	("pratos", 1)	("de", 1)	("trigo", 1)	("para", 1)	("dois", 1)	("tigres", 1)

Com combiners:

mapper1	("um", 2)	("prato", 1)	("de", 1)	("trigo", 1)	("para", 1)	("tigre", 1)
mapper2	("dois", 2)	("pratos", 1)	("de", 1)	("trigo", 1)	("para", 1)	("tigres", 1)

```
        double soma = 0, total=0;
        for(Text salario: values){
            String valores[] = salario.toString().split(",");
            soma+=Double.parseDouble(valores[0]);
            total+=Integer.parseInt(valores[1]);
        }
        if(soma > 0 && total>0){
            double media = soma/total;
            context.write(key, new DoubleWritable(media));
        }
    }
}
```

Até agora não fizemos nada além de remover a premissa assumida pelo reducer de que o número de pares chave/valor recebido é o mesmo número dos dados de input recebido pelo mapper. Para escrever o combiner precisamos herdar a classe Reducer, e implementar seu método reduce.

Listagem 8. Classe CensoCombiner.

```
public class CensoCombiner extends
Reducer <Text, Text, Text, Text> {
    public void reduce(Text key, Iterable<Text> values,
    Context context) throws IOException,
    InterruptedException {
        double soma = 0, total=0;
        for(Text salario: values){
            String valores[] = salario.toString().split(",");
            soma+=Double.parseDouble(valores[0]);
            total+=Integer.parseInt(valores[1]);
        }
        context.write(key, new Text(soma + "," + total));
    }
}
```

Como se pode notar o combiner é muito similar ao reducer, exceto pelo fato de produzir um resultado parcial. Para que um combiner funcione é preciso que ele seja uma transformação equivalente dos dados ao reducer, de tal forma que se removido não interfira em nada no processo e no resultado final. Para que o combiner seja executado é necessário utilizar o método job.setCombinerClassCensoCombiner.class) na classe de configuração comumente denominada Driver (veja edição anterior).

Classes Reducer predefinidas

Assim como para os mappers, a API do Hadoop também provê implementações de reducers prontas para uso. Nesse caso, são dois reducers de contagem de elementos, disponíveis no pacote org.apache.hadoop.mapreduce.lib.reduce:

- LongSumReducer
- IntSumReducer

Ambos possuem a mesma utilidade, diferindo apenas pelo tipo do valor que os mesmos trabalham (IntWritable e LongWritable, respectivamente).

Exemplo prático

Como cenário de uso das funcionalidades descritas neste artigo, serão implementados dois programas MapReduce que analisam todos os resultados de concursos da Mega-Sena realizados desde 1996. Esses dados podem ser obtidos publicamente no site da Caixa Econômica Federal [5]. Esse HTML será o arquivo de entrada para os dois programas MapReduce. Para tal, o mesmo deve ser baixado e enviado para o HDFS (ver artigo anterior sobre como enviar arquivos para o HDFS). É ainda preciso que esse HTML seja interpretado de alguma forma, para que os dois MapReduce possam receber como chave/valor apenas os dados que realmente os interessam, ou seja, as dezenas sorteadas em cada concurso. Com essa finalidade, foram desenvolvidas as classes MegaSenaResultadosInputFormat, MegaSenaResultadosInputSplit e MegaSenaResultadosRecordReader, as quais implementam, respectivamente, InputFormat, InputSplit e RecordReader. Infelizmente, o conceito de como implementar essas classes é um pouco complexo para ser coberto ainda nesta edição, mas os arquivos com o código-fonte devidamente comentado podem ser baixados do site da revista.

O primeiro MapReduce realizará a contagem de quantas vezes cada uma das 60 dezenas foram sorteadas em concursos. Confiando que as seis dezenas de cada concurso serão recebidas como o valor de cada chamada do método map, este exemplo se configura em um clássico caso de contagem de palavras. Neste caso, o MapReduce completo será implementado utilizando-se da classe TokenCounterMapper e IntSumReducer da API do Hadoop apresentadas nas sessões anteriores. Dessa forma, faz-se necessário apenas desenvolver a classe Driver (Listagem 9), onde serão definidas quais as implementações de Mapper e de Reducer serão usadas, bem como o InputFormat customizado.

Listagem 9. Driver de MapReduce de contagem de ocorrência de cada dezena.

```
@Override
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  Job job = new Job(conf);
```

```
  job.setJarByClass(MegaSenaResultsCounterDriver.class);
  //Seta o nome do Job
  job.setJobName("MegaSenaDezenasCounter");
  //Define a classe TokenCounterMapper
  //provida pela API do Hadoop como Mapper
  job.setMapperClass(TokenCounterMapper.class);
  //Define a classe IntSumReducer provida pela
  //API do Hadoop como Reducer
  job.setReducerClass(IntSumReducer.class);
  //Define a classe IntSumReducer provida pela
  //API do Hadoop como Combiner
  job.setCombinerClass(IntSumReducer.class);
  //Define as chaves de saída como tipo Text
  //(ver sessão Tipos de Dados)
  job.setOutputKeyClass(Text.class);
  //Define os valores de saída como IntWritable
  //(ver sessão Tipos de Dados)
  job.setOutputValueClass(IntWritable.class);
  //Seta o InputFormat customizado para
  //interpretar o HTML de resultados da MegaSena
  job.setInputFormatClass(
    MegaSenaResultadosInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);
  //Informa o path do arquivo de entrada, no caso o
  // caminho doHDFS para o HTML de resultados da
  // MegaSena
  FileInputFormat.setInputPaths(job, new Path(args[0]));
  //Informa o path do arquivo de resultado
  FileOutputFormat.setOutputPath(job,
    new Path(args[1]));

    boolean success = job.waitForCompletion(true);

    return (success ? 0 : 1);
}
```

Note que fizemos uso do combiner forçando que os dados sejam combinados ainda na fase de mapper. Neste exemplo em específico de contagem de dezenas obtivemos 6.690 registros de entrada para o combiner (Combine input records) e 240 registros de saída (Combine output records). Ou seja, a quantidade de registros que foi enviado a um reducer foi significativamente menor do que seria se não tivemos feito uso de um combiner. Porém, combiners não necessariamente melhoram a performance, é preciso monitorar o job para ver se o número de registros reduz significativamente a ponto de justificar o tempo de execução extra de um combiner. Faça este acompanhamento através da interface web do jobtracker.

O segundo MapReduce analisará se um mesmo conjunto de 6 dezenas já se repetiu em mais de um concurso. Este seria um cenário de soma de valores ainda mais simples que o exemplo anterior, não fosse pelo fato de os valores apresentarem as dezenas ordenadas por ordem de sorteio. Essa particularidade pode fazer com que um possível resultado como 10, 20, 30, 40, 50, 60 seja contabilizado como sendo diferente de outro concurso tendo como resultado 60, 50, 40, 30, 20, 10. Tal problema pode ser resolvido com a implementação de um Mapper que ordene os números sorteados por ordem crescente de valor, antes de passar o valor para ser contabilizado no Reducer. Novamente será utilizada a InputFormat customizada e o IntSumReducer. Os tipos de dados utilizados em ambos os exemplos são Text e IntWritable. As Listagens 10 e 11, respectivamente, exibem os blocos de código da classe Driver e Mapper deste exemplo contendo a lógica principal do MapReduce. O código completo pode ser baixado do site da revista.

Listagem 10. Driver de MapReduce de contagem de repetição de resultados da Mega-Sena.

```
@Override
public int run(String[] args) throws Exception {
   Configuration conf = getConf();
   Job job = new Job(conf);
   //Seta o nome do Job
   job.setJarByClass(
      MegaSenaResultadosCounterDriver.class);
   job.setJobName("MegaSenaResultadosCounter");
   //Define a classe MegaSenaMapper implementada no
   // exemplo como Mapper
   job.setMapperClass(MegaSenaMapper.class);
   //Define a classe IntSumReducer provida pela
   //API do Hadoop como Reducer
   job.setReducerClass(IntSumReducer.class);
   //Define as chaves de saída como tipo Text
   //(ver sessão Tipos de Dados)
   job.setOutputKeyClass(Text.class);
   //Define os valores de saída como IntWritable
   //(ver sessão Tipos de Dados)
   job.setOutputValueClass(IntWritable.class);
   //Seta o InputFormat customizado para interpretar o
   //HTML de resultados da MegaSena
   job.setInputFormatClass(
      MegaSenaResultadosInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
   //Informa o path do arquivo de entrada,
   //no caso o caminho do HDFS para o HTML de
   //resultados da MegaSena
   FileInputFormat.setInputPaths(job, new Path(args[0]));
   //Informa o path do arquivo de resultado
   FileOutputFormat.setOutputPath(job,
      new Path(args[1]));
   boolean success = job.waitForCompletion(true);
   return (success ? 0 : 1);
}
```

Listagem 11. Mapper de contagem de repetição de resultados da Mega-Sena.

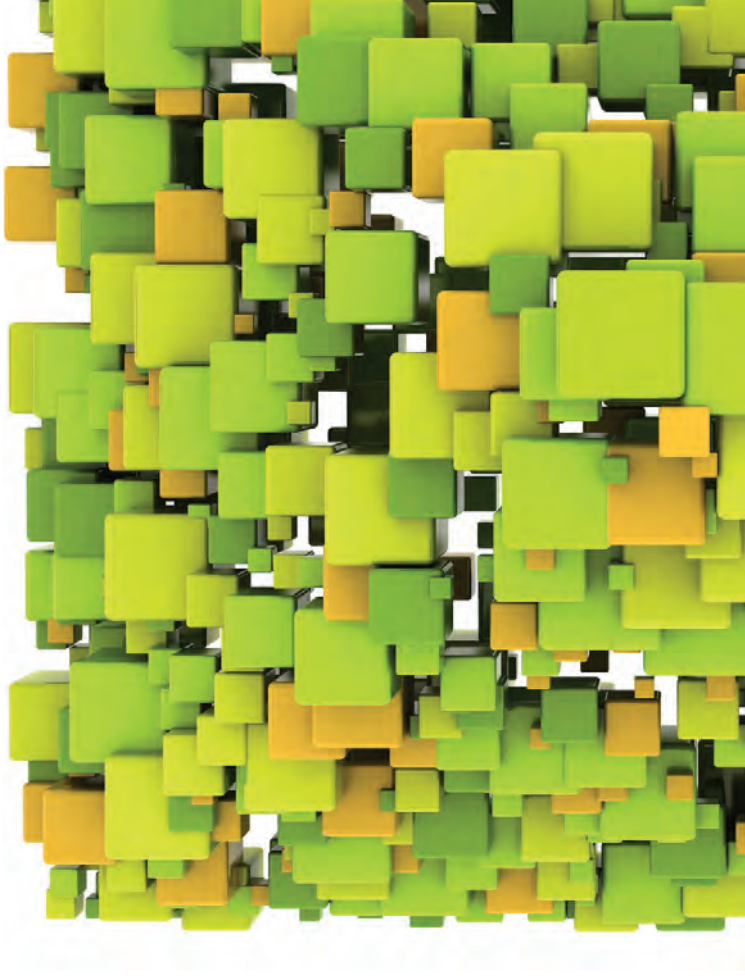
```
@Override
public void map(Text key, Text value, Context context)
throws IOException, InterruptedException{

    List<String> sortedNumbers = new ArrayList<String>();
    //faz parse do valor por \t e adiciona cada dezena na
    // lista
    for(String number : value.toString().split("\t"))
        sortedNumbers.add(number);
    //orderna as dezenas em ordem crescente
    Collections.sort(sortedNumbers);
    //concatena novamente as dezenas em um único valor,
    //agora em ordem numérica crescente
    StringBuilder builder = new StringBuilder();
    for(String number : sortedNumbers)
        builder.append(number + "\t");
    //escreve o valor (resultado do sorteio) para o Reducer,
    //e associa um valor unitário. No Reducer, serão
    // somadosos valores unitários para os resultados
    // repetidos
    context.write(new Text(builder.toString()),
        new IntWritable(1));
}
```

Considerações finais

O presente artigo detalhou o funcionamento do MapReduce, começando por uma análise de como o Hadoop disponibiliza os dados de entrada para os jobs, o que acontece entre as fases de map e reduce, como o conteúdo é formatado em chave e valor e como o processamento é organizado em um resultado. Em seguida, foi apresentado todo um conjunto de classes da API do Hadoop, úteis na implementação de programas MapReduce, onde são, frequentemente, reutilizadas por completo, ou, em alguns cenários como o do exemplo prático citado, customizadas para um cenário específico.

Apesar de introdutório e básico, os conceitos aqui apresentados capacitam o desenvolvedor a tomar decisões sobre o formato dos dados a se utilizar, uso de classes predefinidas, a utilização de técnicas que aumentam o volume de processamento em detrimento do consumo de banda ou vice-versa e quando utilizá-las. Além de demonstrar o poder da customização de classes padrões do hadoop com o exemplo prático.

Existem ainda outras APIs a serem detalhadas em artigos futuros, como a do HDFS, útil em cenários de uso de integração com o sistema de arquivos do Hadoop, ou ferramentas adicionais do ecossistema Hadoop, como o projeto Mahout da Apache, que disponibiliza uma série de algoritmos de Aprendizagem de Máquina (Machine Learning) implementados como MapReduce para o Hadoop, e o HBase, um banco de dados NoSQL distribuído totalmente integrado ao Hadoop.



/para saber mais

Na edição 52 da MundoJ foi publicado o artigo "Processando Dados com Hadoop", apresentando uma introdução sobre esta tecnologia. O atual artigo aproveita o exemplo de censo mostrado anteriormente para demonstrar a utilização de recursos adicionais de otimização de desempenho da API de MapReduce.

/referências

> Wiki do Hadoop 0.20.2: http://cloud.ozyegin.edu.tr/Hadoop-UML-Diagrams/Documentation/html/index.html

> Hadoop API Docs: http://hadoop.apache.org/common/docs/current/api/overview-summary.html

> Livro Hadoop In Action, de Chuck Lam

> Música Etnia de Chico Science e Nação Zumbi usada na Listagem 2

> Livro Hadoop, The Definitive Guide, de Tom White

> Caixa Econômica Federal, arquivo com os resultados da Mega-Sena em formato HTML http://www1.caixa.gov.br/loterias/loterias/megasena/download.asp