UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
TRABALHO DE CONCLUSÃO DE CURSO

Um framework para MapReduce em clusters

Gabriel Moreno Frota

Banca examinadora

Prof. Elvio J. Leonardo – (orientador)

Prof. Edson Alves de Oliveira Junior

Prof. Lucas Pupulin Nanni

Maringá, 26 de maio de 2021

Um framework para MapReduce em clusters

Trabalho de Conclusão de Curso apresentado à Universidade Estadual de Maringá, como parte dos requisitos necessários à obtenção do título de Bacharel em Informática.

Orientador: Prof. Dr. Elvio J. Leonardo

Maringá, 26 de maio de 2021

Sumário

1 Introdução

Com a presença cada vez maior da internet no mundo, programadores em variadas situações necessitam implementar uma determinada computação para a análise de dados *big data*, tal como documentos de texto da web, logs de servidores, dados estruturados em bancos de dados, etc. Essas computações são, de uma maneira geral, conceitualmente simples. Entretanto, o volume de dados de entrada tende a ser muito grande e, para que a computação termine em um tempo tolerável, é necessário que seja distribuída entre várias máquinas. O problema de como distribuir os dados e a computação, de como agregar o resultado final e de como tratar possíveis falhas de hardware ou de rede, tende a aumentar muito a complexidade do problema originalmente simples.

Em resposta a isso, pode-se trabalhar com uma abstração mais simples, que permita expressar o problema original da análise dos dados e esconda a complexidade da distribuição do trabalho em um *cluster*, por meio de um *framework*. O *framework* proposto é inspirado nas funções **map** e **reduce**, originadas na linguagem Lisp, e presentes em muitas outras linguagens. O modelo de programação MapReduce funciona bem para esse tipo de problema, pois tais computações podem ser implementadas pela aplicação de uma espécie de **map** em cada elemento da entrada, geração de dados intermediários no formato de registro [chave, valor], seguidos pela aplicação de uma espécie de **reduce** em todos elementos de chave igual, para agregação do resultado final.

O usuário do *framework*, então, implementa algumas funções e o sistema se encarrega dos detalhes da distribuição dos dados de entrada, paralelização da computação, tratamento de possíveis falhas e agregação do resultado final. Isso permite que programadores sem conhecimentos avançados de sistemas distribuídos possam utilizar os recursos de um *cluster,* de uma maneira relativamente simples.

1.1 Motivação e Justificativa

O modelo de computação distribuída descrito acima tornou-se muito popular no mundo nas últimas décadas. MapReduce do Google foi o pioneiro, o qual foi usado no sistema indexador gerador das estruturas de dados que são usadas no serviço de busca [1]. Sistemas como Dryad [2], Map-ReduceMerge [3], e Spark [4], incrementaram as ideias iniciais, particularmente no que diz respeito ao fluxo dos dados e o modelo de execução. MapReduce tem um modelo de execução bastante rígido, que é uma vantagem quanto à facilidade de uso, porém resulta em um desempenho ruim em certos tipos de problemas que não se encaixam bem nesse modelo [4]. O sistema nesses casos acaba realizando muito trabalho desnecessário, que poderia ser evitado caso o programador tivesse mais controle sobre a execução da tarefa [4]. Esse é o problema chave que os sistemas mais recentes tentam resolver: oferecer um controle mais fino sobre a execução da tarefa, para obtenção de um melhor desempenho em um conjunto maior de problemas, mantendo sempre em mente a facilidade de uso da ferramenta, pois algo muito difícil de se usar perde a ideia original de facilitar a vida do programador. O último grande avanço da área foi a ferramenta Spark [4], que surgiu na universidade de Berkeley, no laboratório AMPLab, e é hoje uma das ferramentas de processamento distribuído mais utilizadas no mundo [5].

1.2 Objetivos

Este trabalho tem como objetivo a implementação de um *framework* para processamento distribuído no modelo MapReduce, que será uma versão simplificada do Hadoop [6] com algumas ideias originais do autor. O *framework* é em sua essência um conjunto de abstrações que permitem expressar e executar uma computação distribuída de uma maneira simples e produtiva.

1.3 Metodologia

Hadoop é um projeto muito extenso, e o tempo para o trabalho é limitado, portanto uma versão menor que capture as ideias centrais da tecnologia é a intenção do trabalho. O aluno acredita que Hadoop contém um conjunto de ideias importantes, que estão presentes em muitas tecnologias de sucesso na era da internet, além de ser um problema complexo de engenharia de software, portanto entende-se que a implementação de um projeto desse tipo será um aprendizado de conhecimentos valiosos para o futuro. Foi usado como material de consulta para o trabalho, o livro Hadoop The Definitive Guide [6]. As figuras do documento onde a fonte não é indicada foram produzidas pelo autor. As simplificações do projeto de uma maneira geral são:

1→ A entrada e saída de Hadoop MapReduce é uma lista de arquivos, e no projeto do trabalho é um único arquivo de entrada e um único arquivo de saída.

2→ Hadoop MapReduce tem intenção de ser executado "em cima" de um sistema de arquivos distribuídos, em arquivos de entrada e saída que estão quebrados em blocos espalhados no *cluster*. É papel do sistema de arquivos informar quais máquinas contém os blocos da entrada, nas quais terão as etapas MapReduce executadas pelo *framework*, seguidas da escrita da saída também espalhadas em blocos nas diferentes máquinas. Ou seja, não é responsabilidade do MapReduce o estado da entrada e da saída, isso é responsabilidade do sistema de arquivos distribuídos, e entrada e saída encontram-se ambas espalhadas nas diferentes máquinas do *cluster*. No projeto do trabalho isso é diferente, pois o aluno não quer que o projeto dependa de um sistema de arquivos previamente instalado e configurado para a execução, além de ter entrada e saída simplificadas para um arquivo apenas. No projeto do trabalho são etapas da tarefa a quebra da entrada em blocos e distribuição na rede no início, e a junção dos blocos da saída em um único arquivo no fim.

3→ A execução de uma tarefa Hadoop MapReduce consiste do envio do programa MapReduce ao programa "gerente do *cluster*", que aloca recursos e coordena o andamento da execução. No projeto do trabalho isso é diferente, pois o aluno não quer que o projeto dependa de uma aplicação previamente instalada e configurada para a execução. No projeto do trabalho, os executáveis *Master* e *Worker* são mais ou menos o programa "gerente do *cluster*", capturando suas ideias centrais, no entanto de uma maneira menos flexível que requer menos configurações da parte do usuário.

4 → Hadoop MapReduce é capaz de se recuperar de falhas de máquinas individuais durante a execução da tarefa. O "gerente do *cluster"* percebe que uma máquina morreu (não responde), e precisa então por meio do sistema de arquivos distribuídos, descobrir uma nova máquina que tem os mesmos blocos dos arquivos que a máquina que morreu tinha. A nova máquina então assume o mesmo "pedaço da execução" que era responsabilidade da máquina que morreu, e a execução da tarefa como um todo segue em frente. No projeto do trabalho isso é diferente, pois apesar de o aluno inicialmente planejar a implementação de algo similar, o tempo para a realização do trabalho foi insuficiente. O projeto do trabalho em uma situação de falha de uma máquina abortará a execução da tarefa e retornará uma mensagem de erro no console. A execução inicialmente ficará bloqueada aguardando algum retorno de uma máquina que morreu. Entretanto eventualmente desbloqueará e retornará erro em algum *timeout* da biblioteca Java, abortando a execução do programa e imprimindo o erro.

5→ Hadoop MapReduce possui grande quantidade de parâmetros de execução com intenção de otimização da performance, e tem também um vasto conjunto de possíveis formatos de arquivos de entrada ou saída já implementados. No projeto do trabalho isso é diferente, pois algo menor devido ao tempo é necessário. O projeto do trabalho tem poucos parâmetros de execução, e o *framework* sabe lidar por padrão apenas com arquivos de texto UTF-8.

6→ Planejou-se inicialmente que com o desenvolvimento do projeto concluído, o aluno coletaria estatísticas de execução das tarefas exemplificadas na seção 4.1 com um *profiler*, em uma rede de alta qualidade com várias máquinas de boa performance no DIN, e escreveria uma seção

com alguns gráficos e tabelas relacionadas a isso após a seção 5.6. No entanto, a pandemia de Covid-19 se estendeu por muito tempo e estes planos foram abandonados, pois não é possível frequentar normalmente a UEM. O aluno usou para o desenvolvimento do projeto, 2 *notebooks* velhos e um roteador *wireless* comum formando a rede, que foram suficientes para testar o projeto em relação a corretude da execução. No entanto, uma seção de avaliação da performance baseada em execuções nestas poucas máquinas velhas não seria representativa do potencial do trabalho. A coleta de estatísticas e avaliação da performance são tarefas para investigações futuras.

2 Arquitetura do framework

O *framework* é implementado na linguagem Java, consiste de classes que modelam a tarefa a ser executada, e de dois executáveis que constituem o ambiente de execução. O ambiente de execução é composto de um executável *Master* e um executável *Worker*, onde um processo *Master* coordena a execução de uma tarefa distribuída entre N processos *Workers*. Ambos são programas com interface de linha de comando, que utilizam a tecnologia Java RMI *(Remote Method Invocation)* para comunicação entre os processos nas diferentes máquinas. A Figura 1 descreve a relação entre os executáveis e os valores das portas importantes.

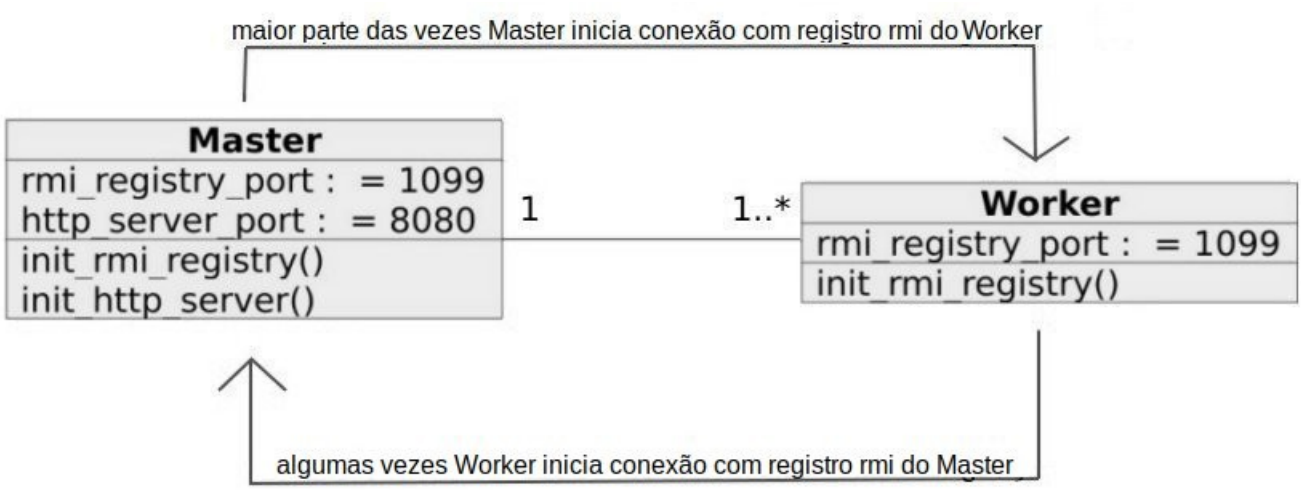

Figura 1: Executáveis que constituem o ambiente de execução

Ambos processos inicializam o registro RMI, pois ambos assumem o papel de servidor ou cliente, em diferentes etapas da execução. O *Master* também inicializa um servidor HTTP, pois a classe que contém o código que será executado nos *Workers* será enviada por esse servidor. O *Master* encapsula uma única execução de uma única tarefa, distribuindo e coordenando o trabalho por meio da rede. O *Worker* é uma espécie de servidor que inicializa e executa por tempo indeterminado, recebendo do *Master* o código a ser executado. Cada executável está presente em uma máquina diferente, utilizando a porta 1099 para comunicação RMI. O *Master* recebe de entrada os parâmetros da execução por meio de arquivos e possíveis *flags* de configuração, como indicado na Figura 2.

```
Usage: exec/Master [-ov] [-bs=INT_VAL] -i=FILE_PATH -mr=FILE_PATH -o=FILE_PATH
                   -w=FILE_PATH
Master process for MapReduce.
 -bs, --buffersize=INT_VAL  In-memory Record buffer size.
 -i,  --input=FILE_PATH     Input file path.
 -mr, --mapreduce=FILE_PATH MapReduce implementation file path.
 -o,  --output=FILE_PATH    Output file path.
 -ov, --overwrite           Overwrite split files in Workers.
 -w,  --workers=FILE_PATH   Workers IP addresses file path.
```

Figura 2: Mensagem de ajuda do *Master*

- **-bs:** inteiro que significa o tamanho do *array* alocado para uma partição em memória no *PartitionWriter.* Este valor controla quantos *Records* mantêm-se em memória antes de dar *spill*. As classes que utilizam o valor deste parâmetro serão descritas na seção 5.6.
- **-i:** O *framework* espera ler um arquivo existente a partir deste caminho, que serão os dados de entrada para a etapa **map**.
- **-mr:** O *framework* espera ler um arquivo que é uma classe que estende MapReduce a partir desse caminho. O funcionamento deste parâmetro será descrito na seção 5.
- **-o:** O *framework* escreve a saída da etapa **combine** no arquivo descrito por este caminho.
- **-ov:** Esta *flag* presente na chamada faz com que o *Master* sobrescreva um *split file* da entrada já existente em algum *Worker* com um novo envio e escrita. A ausência da *flag* faz com que o *Master* não envie um *split file* de nome repetido a um *Worker.* O uso do valor deste parâmetro será descrito na seção 5.2.
- **-w:** O *framework* espera ler um arquivo de texto a partir deste caminho, o qual contém os endereços IPs dos *Workers* que participarão da execução da tarefa. É esperado um IP por linha do texto.

O *framework* é usado pelo usuário por meio de um arquivo JAR (*Java Archive*), do qual pode-se executar um processo *Master* ou um processo *Worker,* ou estender classes para a implementação de uma tarefa para execução. O código do projeto está disponível no github, através do endereço https://github.com/GabrielFrota/MapReduce. A Figura 3 contém a estrutura do projeto e uma breve descrição de cada classe. A Figura 4 ilustra os pacotes e as classes do projeto, detalhando também as relações por meio de um diagrama UML (*Unified Modeling Language*).

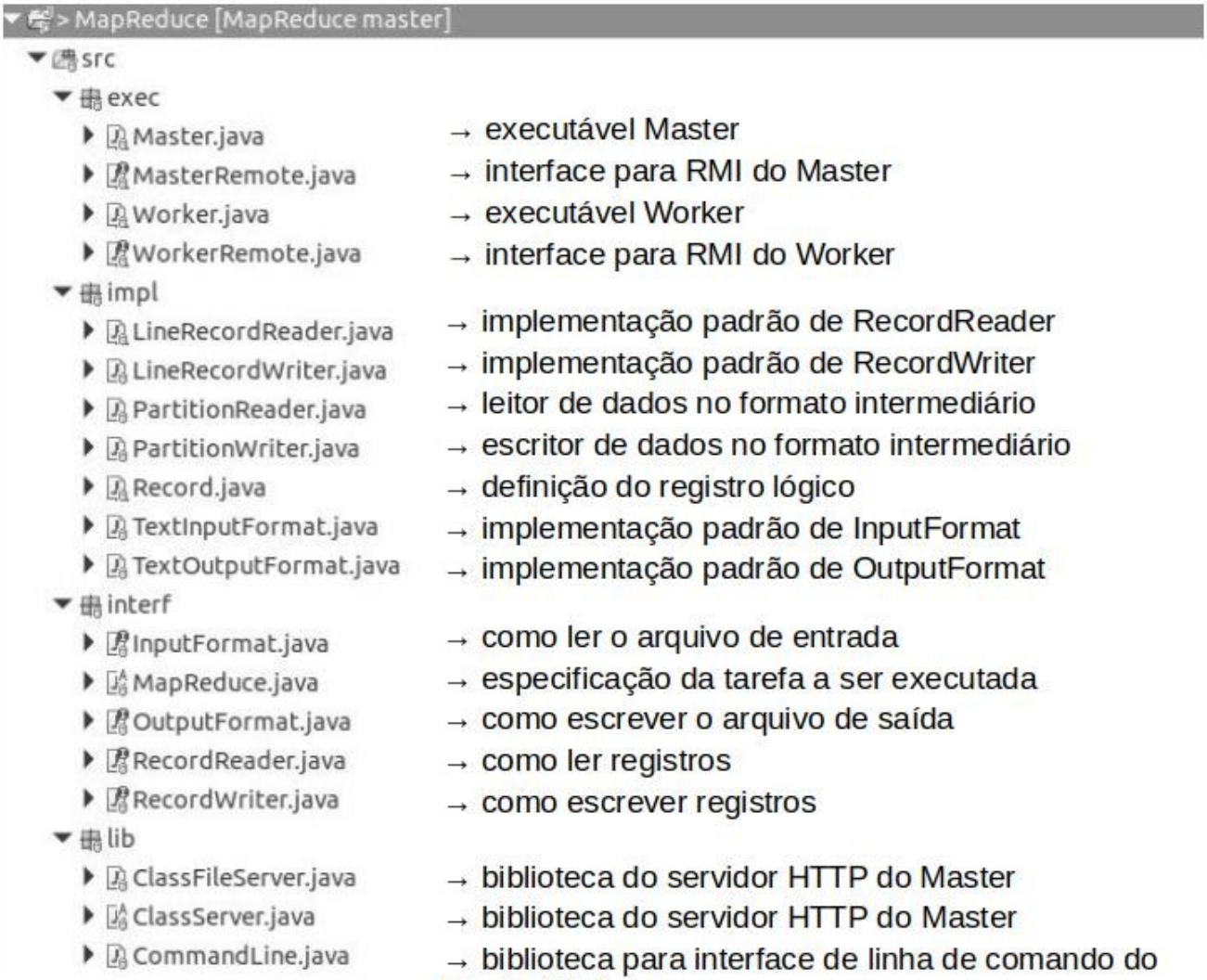

Figura 3: Estrutura do projeto

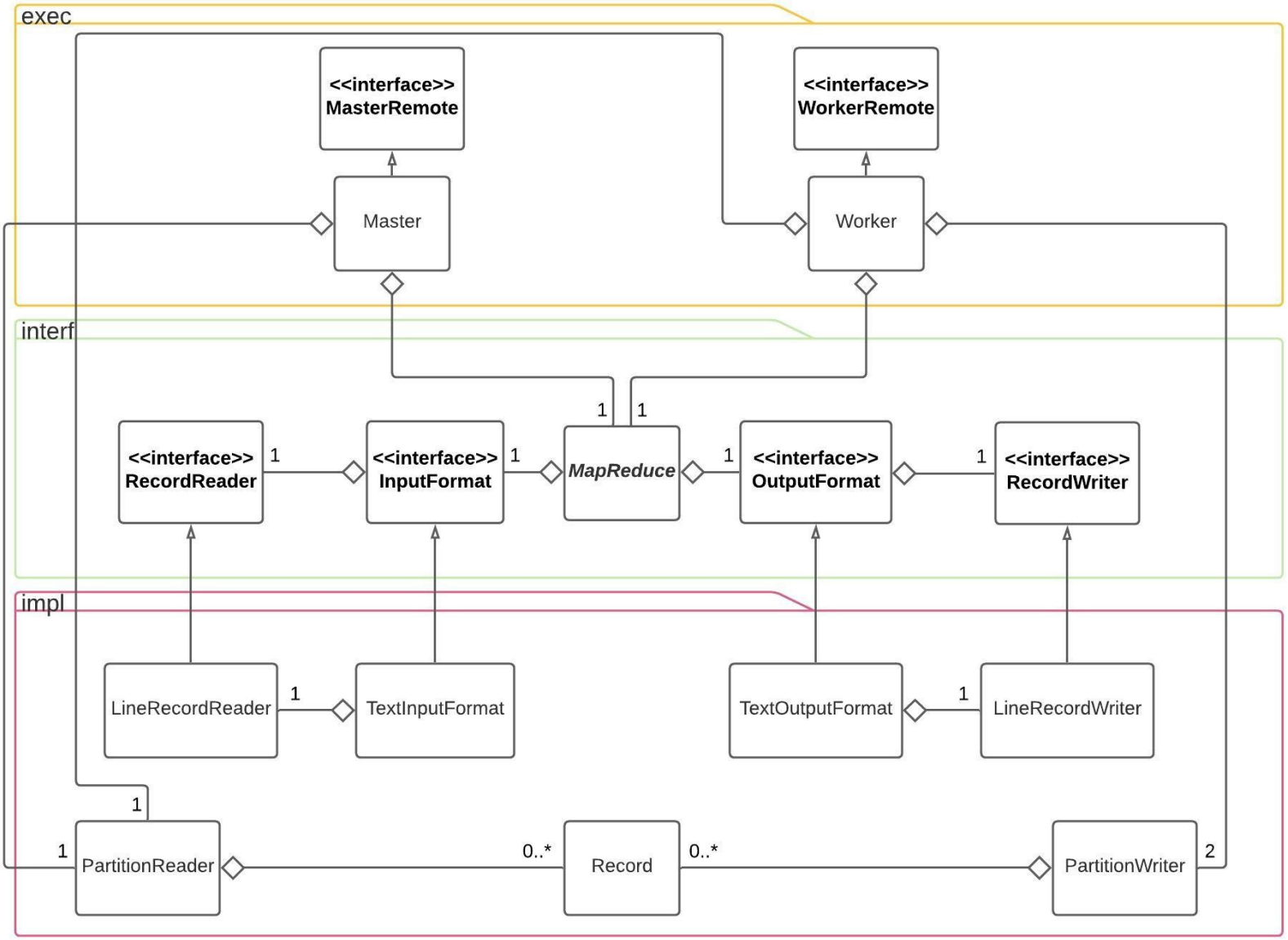

Figura 4: Classes do *framework* e suas relações

3 Java Remote Method Invocation

Java RMI (*Remote Method Invocation*) é a solução padrão para RPC (*Remote Procedure Call*) no ambiente Java. Tem a intenção de facilitar a execução de métodos em JVMs (*Java Virtual Machine*) diferentes, potencialmente executando em máquinas diferentes [8]. É utilizado extensivamente pelo *framework* na comunicação entre processos, portanto a compreensão de seu funcionamento é importante. A Figura 5 mostra uma noção de uma "JVM cliente" requisitando a execução de um método em uma "JVM servidor", bloqueando a *thread* na "JVM cliente" até a resposta da "JVM servidor".

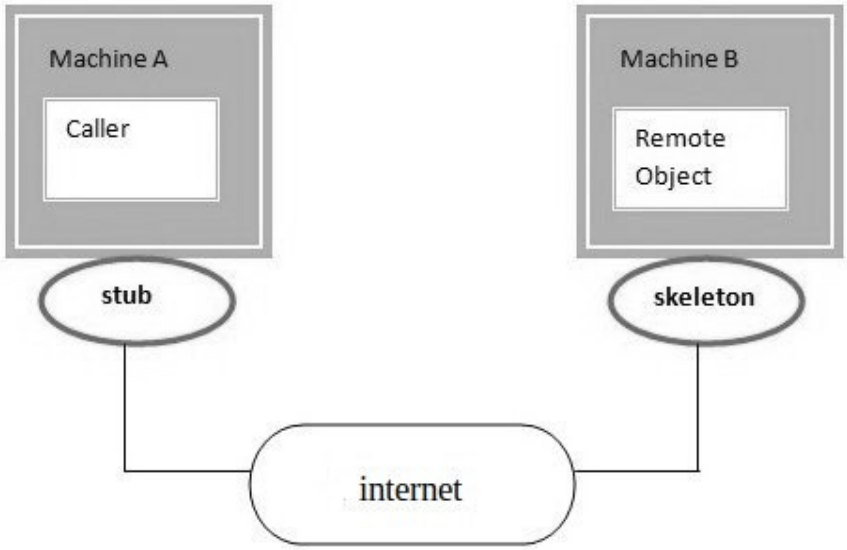

Figura 5: Invocação remota de método. Fonte: www.javatpoint.com/RMI, acesso 19/05/2021

stub é a abstração do objeto remoto do lado cliente, e um trecho de código ao chamar um de seus métodos, terá os seguintes passos executados:

1- Inicialização de uma conexão com lado servidor;
2- Transmissão do método a ser executado e o valor dos parâmetros;
3- Bloqueio até o recebimento da resposta do lado servidor;
4- Processamento da resposta, que pode ser uma resposta válida ou exceção indicando erro;
5- Retorno dos dados para o trecho de código que chamou o método;

O trecho a seguir pertencente ao *Master*, inicializa um s*tub* do tipo *WorkerRemote* por meio do identificador *WorkerRemote.NAME* e executa um de seus métodos. Nessa situação o *Master* é o cliente, e o *Worker* o servidor. O método *createNewFile* será executado no *Worker*, enquanto a *thread* em execução no *Master* bloqueia até o recebimento do retorno.

```
private WorkerRemote getWorkerRemote(String ip)
  throws RemoteException, AccessException, NotBoundException
{
  var reg = LocateRegistry.getRegistry(ip);
  return (WorkerRemote) reg.lookup(WorkerRemote.NAME);
}

var worker = getWorkerRemote(ip);
worker.createNewFile(mapRed.getInputName());
```

skeleton é a abstração do objeto remoto do lado servidor, que ao receber uma requisição RMI válida, tem os seguintes passos executados:

1- Leitura do método a ser executado e o valor dos parâmetros;
2- Invocação do método no contexto do objeto;
3- Transmissão do retorno do método ao lado cliente;

O trecho a seguir pertencente ao *Worker,* inicializa um *skeleton* do tipo *WorkerRemote* na JVM em execução. O código mapeia o objeto **impl** no registro RMI com o identificador *WorkerRemote.NAME*. Com isso a JVM agora é um servidor de *WorkerRemote*, e executará métodos no objeto **impl** ao receber requisições RMI válidas, vindas de um cliente com um *stub WorkerRemote*.

```
var impl = new WorkerRemoteImpl();
var reg = LocateRegistry.createRegistry(1099);
reg.bind(WorkerRemote.NAME, impl);
```

Observa-se então que com o RMI, o programador está sempre lidando com objetos Java no contexto de uma JVM, e a tecnologia automatiza o processo muito trabalhoso de gerência de conexões, leitura e escrita dos dados em *sockets*, serialização e deserialização de objetos, tratamento de erros, etc.

4 Modelo de execução da tarefa

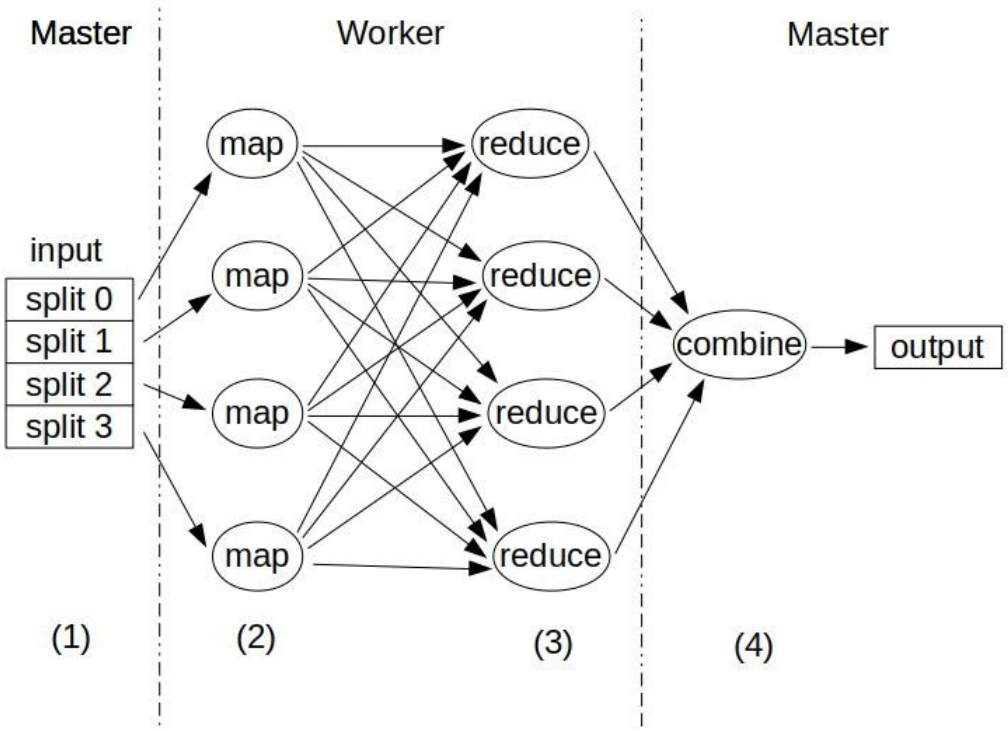

Figura 6: Visão geral da execução

A execução de uma tarefa consiste da execução das seguintes etapas, onde os rótulos numerados na Figura 6 correspondem aos itens da lista:

1→ Divisão do arquivo de entrada (parâmetro **-i**) em N *splits* pelo *Master*, onde N é a quantidade de *Workers* na tarefa (quantidade de IPs no parâmetro **-w**), seguido da transmissão de um *split* para cada máquina *Worker.*

2→ Execução da etapa **map** em paralelo nas diferentes máquinas *Worker*, seguidos da divisão dos dados intermediários gerados pela função em N partições, onde cada partição é um *split* do espaço das chaves intermediárias, computados através da operação *(hash(chave)* ***mod*** *N)*.

3→ Execução da etapa **reduce** em paralelo nas diferentes máquinas *Worker*, seguidos da transmissão dos dados intermediários gerados pela função ao *Master.*

4→ Execução da etapa **combine** no *Master,* que é um **reduce** que tem de entrada a saída dos **reduces** de todos os *Workers.* A intenção dessa etapa é combinar os *splits* da computação em um único arquivo, onde o usuário pode apenas concatenar os dados de uma maneira direta em casos simples, ou potencialmente realizar uma operação mais complexa, que reduz todos os dados a um único valor.

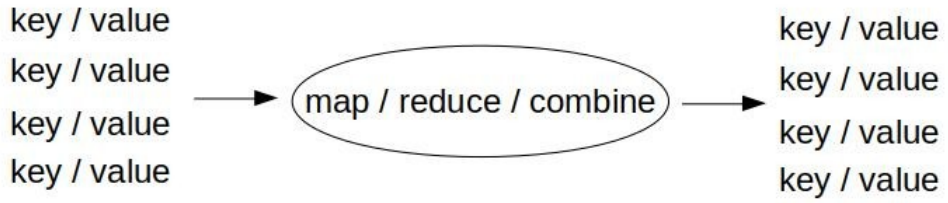

Figura 7: Formato dos dados

Figura 7 ilustra a ideia de que a computação tem como entrada uma sequência de registros com chave/valor, e tem como saída uma sequência de registros com chave/valor. Os tipos da saída são independentes dos tipos da entrada, e os tipos da entrada são os tipos da saída da etapa anterior. Isso é uma forma abstrata de ver os dados pelo *framework*, que é implementado com um código genérico, e é papel do usuário especificar os tipos concretos para cada etapa da computação. Esse processo será visto em mais detalhes na seção 5.1.

Adiante alguns exemplos de problemas que são facilmente expressados como tarefas MapReduce.

URL mais requisitada: A etapa **map** processa logs de servidores e emite na saída [URL, 1]. A etapa **reduce** soma todos os valores de mesma URL e emite na saída [URL, total]. A etapa **combine** emite na saída o [URL, total] de maior total.

Grep distribuído: A etapa **map** processa o documento e emite na saída a linha que contém o padrão especificado. A etapa **reduce** emite a entrada na saída, encaminhando os dados exatamente iguais para a próxima etapa. A etapa **combine** emite a entrada na saída, concatenando os *splits* da computação em um único arquivo.

Índice invertido: A etapa **map** processa o documento e emite na saída [palavra, offset_no_documento]. A etapa **reduce** agrupa todos os *offsets* de palavra igual e emite na saída [palavra, lista(offset_no_documento)]. A etapa **combine** emite a entrada na saída, concatenando os *splits* da computação em um único arquivo, formando um índice invertido do documento.

Ordenação distribuída: A etapa **map** extrai a chave de cada registro da entrada e emite na saída [chave, registro]. A etapa **reduce** emite a entrada na saída, encaminhando os dados exatamente iguais para a próxima etapa. A etapa **combine** emite a entrada na saída, concatenando os *splits* da computação em um único arquivo agora ordenado. Isso funciona por causa do mecanismo que será visto na seção 5.6, onde os dados das etapas intermediárias são ordenados pela chave.

Grafo Web-Link reverso: A etapa **map** processa páginas web e emite na saída [URL_destino, URL_origem] a cada link para uma URL_destino vistos em uma página de URL_origem. A etapa **reduce** agrega e concatena em uma lista, todas as URL_origem que possuem um link para URL_destino e emite na saída [URL_destino, lista(URL_origem)]. A etapa **combine** emite a entrada na saída, concatenando os *splits* da computação em um único arquivo.

5 Execução da tarefa

O usuário do *framework* precisa implementar a computação a ser executada, que é modelada por algumas interfaces e a classe abstrata MapReduce, a qual possui métodos com intenção de serem sobrescritos pelo usuário. O *framework* chama cada um desses métodos em momentos definidos da execução, portanto é preciso que o usuário entenda a execução da tarefa para saber como usá-los. O *framework* recebe essa implementação através do parâmetro obrigatório **-mr** no *Master*, o qual deve ser um caminho para um arquivo que é uma classe que estende MapReduce, no qual assume-se que o usuário implementou seu algoritmo. O que o *Master* faz com esse arquivo é um pouco complicado, e será descrito adiante. A Figura 8 ilustra essa importante etapa na inicialização, onde um *Worker* faz o download e carrega na JVM a classe que contém o código a ser executado.

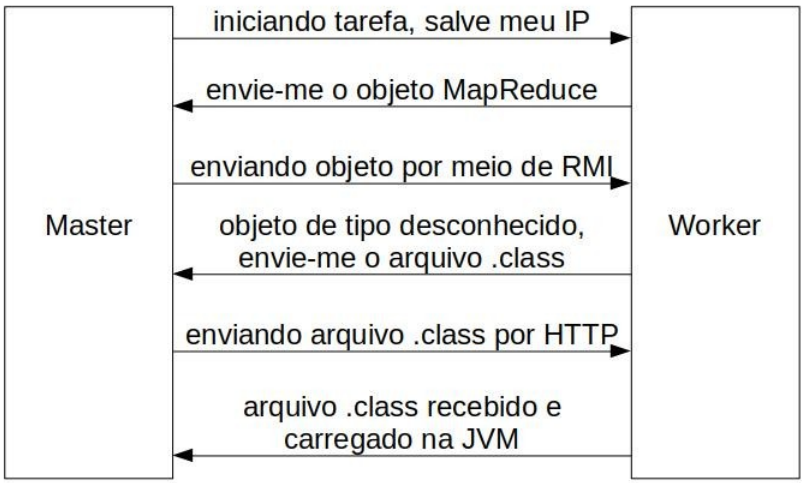

Figura 8: Download da classe MapReduce

No contexto de uma comunicação RMI, quando um cliente recebe do servidor um objeto de uma classe desconhecida, ou seja, a “JVM cliente” não possui uma classe definida com o mesmo nome da classe do objeto recebido, encontra-se uma situação errônea que precisa ser tratada. É necessário que através da propriedade "java.rmi.server.codebase", na qual assume-se conter uma URL, o cliente consiga fazer o download do arquivo *.class* da classe em questão. Caso contrário, a execução para e retorna erro, pois o cliente não é capaz de lidar com o objeto recebido. No contexto do *framework*, o *Worker* é uma espécie de servidor que inicializa e executa por tempo indeterminado, participando da execução de diferentes tarefas, sempre de dentro de uma mesma execução da JVM. No contexto da JVM, o que identifica uma classe é o seu nome, e não existe uma forma da aplicação Java “descarregar (*unload*)” uma classe previamente carregada durante a execução. A Figura 9 ilustra o problema em questão, que é a causa do tratamento complexo do parâmetro **-mr** no *Master.*

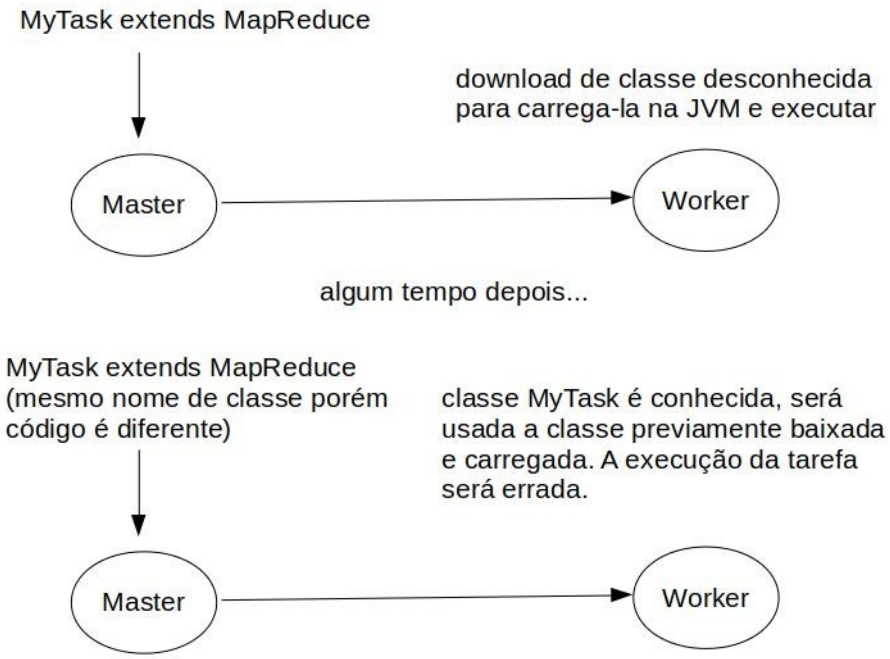

Figura 9: Nome da classe repetido causará execução errada

É preciso garantir que em toda nova tarefa a ser executada, a classe que vem por meio do parâmetro ***–mr*** seja uma classe com nome nunca visto pelas JVMs dos *Workers*, para forçá-las a fazer o download do arquivo atual sempre, e tornar impossível que a situação ilustrada na Figura 9 ocorra. A ideia para a solução utiliza a biblioteca ASM [7], que é um manipulador de *bytecode* Java.

```
String timestamp = "_" + LocalDateTime.now().toString().replace(".", "_");
```

A variável **timestamp** é inicializada no *Master,* e contém uma *string* que expressa o "instante de agora". Com a biblioteca ASM, *Master* gera em memória uma nova classe que é uma cópia da classe vinda de entrada no parâmetro **–mr**, e faz uma travessia em sua árvore sintática substituindo toda ocorrência do nome por **timestamp**. Com isso, a nova classe em memória tem nome **timestamp**, no entanto seu conteúdo executável continua idêntico. *Master* escreve essa classe em memória em um arquivo de nome **timestamp.class**, o qual será disponibilizado para download pelo servidor HTTP *ClassFileServer* na porta 8080, que é informado ao *Worker* de sua existência através da propriedade "java.rmi.server.codebase". *Master* carrega a classe **timestamp** em sua JVM e chama seu construtor, e o objeto que encapsula a execução da tarefa será do tipo **timestamp**, que será sempre um objeto de tipo desconhecido aos *Workers*, garantindo que o download do arquivo atual será feito. A Figura 10 ilustra as etapas da execução de uma forma mais detalhada, destacando algumas interfaces importantes do *framework*.

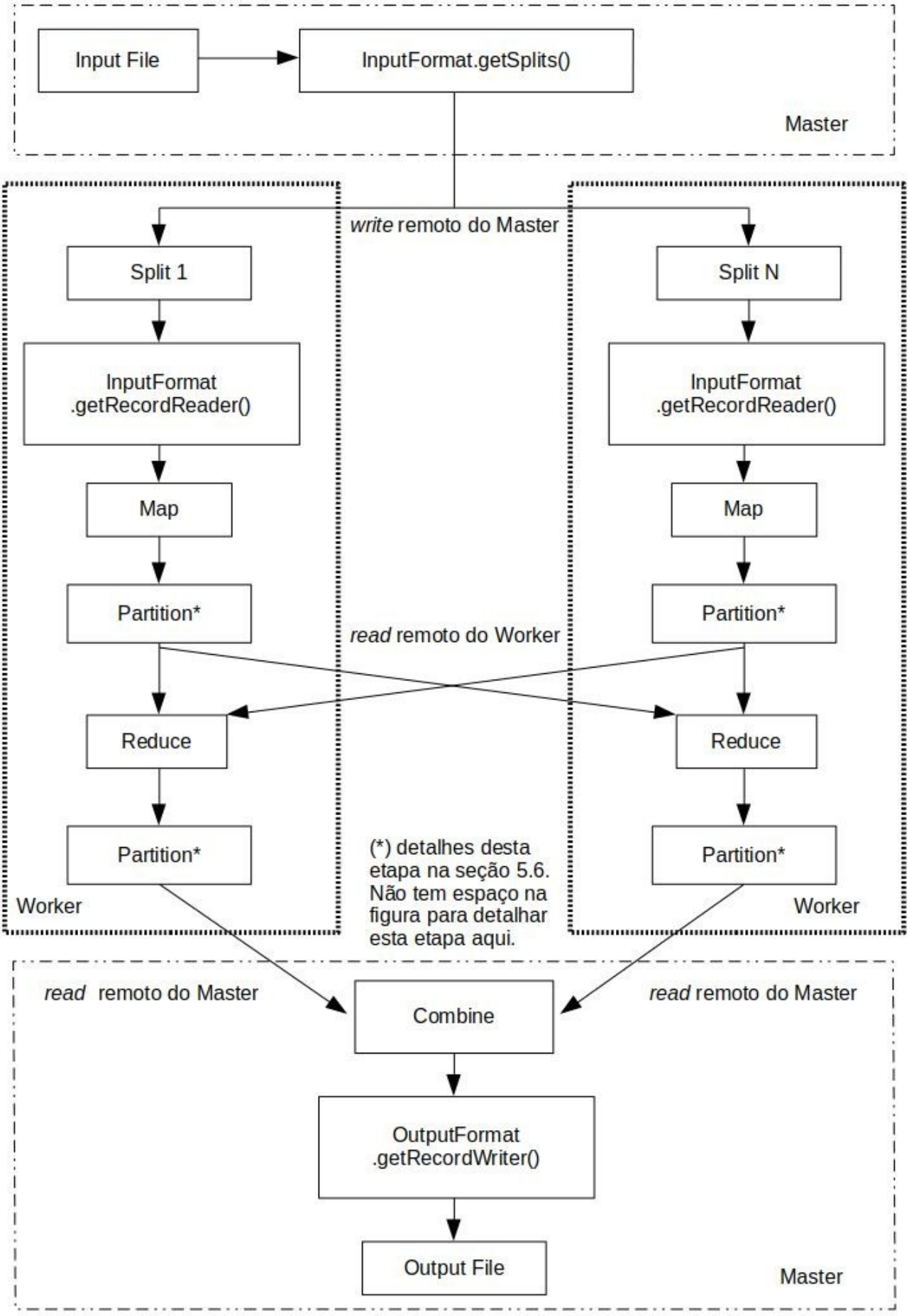

Figura 10: Execução da tarefa e interfaces importantes

5.1 Classe MapReduce

```
public abstract class MapReduce
 <K1, V1,
  K2 extends Comparable<K2> & Serializable, V2 extends Serializable,
  K3 extends Comparable<K3> & Serializable, V3 extends Serializable,
  K4, V4>
    implements Serializable
{
  public abstract InputFormat<K1, V1> getInputFormat();

  public abstract OutputFormat<K4, V4> getOutputFormat();

  public abstract void map(K1 k, V1 v, RecordWriter<K2, V2> w)
    throws IOException;

  public abstract void reduce(K2 k, Iterable<V2> v, RecordWriter<K3, V3> w)
    throws IOException;

  public abstract void combine(K3 k, Iterable<V3> v, RecordWriter<K4, V4> w)
    throws IOException;

  public void preMap(RecordWriter<K2, V2> w) throws IOException {};
  public void postMap(RecordWriter<K2, V2> w) throws IOException {};
  public void preReduce(RecordWriter<K3, V3> w) throws IOException {};
  public void postReduce(RecordWriter<K3, V3> w) throws IOException {};
  public void preCombine(RecordWriter<K4, V4> w) throws IOException {};
  public void postCombine(RecordWriter<K4, V4> w) throws IOException {};
}
```

– Classe MapReduce encapsula a tarefa a ser executada. O usuário implementa sua tarefa sobrescrevendo esta classe, e a executa por meio do parâmetro **-mr** no *Master*.
- K1 V1 são os tipos da entrada da etapa **map** que são os dados lidos do arquivo de entrada.
- K2 V2 são os tipos da saída da etapa **map** e entrada da etapa **reduce**.
- K3 V3 são os tipos da saída da etapa **reduce** e entrada da etapa **combine**.
- K4 V4 são os tipos da saída da etapa **combine** que são os dados escritos no arquivo de saída.
- Todo tipo que será enviado na rede por RMI precisa implementar *Serializable*, pois é preciso uma padronização de como traduzir o objeto para uma sequência de bytes.
- Os tipos das chaves K2 e K3 precisam implementar *Comparable*, pois os dados intermediários das etapas **reduce** e **combine** são ordenados pela chave.
- *InputFormat* e *OutputFormat* são a definição do processamento sobre os arquivos de entrada e saída da tarefa. A ideia é de que o conteúdo e formato dos arquivos é desconhecido ao *framework*, que vê os dados na execução como uma sequência de registros. É papel do usuário especificar o algoritmo que lida com os arquivos de uma tarefa em particular.
- Método *getInputFormat* retorna um objeto que sabe quebrar o arquivo de entrada em N *splits*, e ler seus dados no formato registro [K1, V1]. O *framework* usa o retorno desse método para processar a entrada.
- Método *getOutputFormat* retorna um objeto que sabe escrever registro [K4, V4] no arquivo de saída. O *framework* usa o retorno desse método para escrever o arquivo de saída da tarefa.
- Os métodos (**map, reduce, combine**) são a computação que será executada no elemento (**k, v**) e terá saída escrita por (**w**). O que exatamente é (**k, v, w**) varia conforme a etapa da execução, onde (**k, v**) são a chave e valor de tipos especificados pelo usuário, e (**w**) é um objeto que sabe escrever a

saída no formato correto para a próxima etapa. As 3 etapas são executadas num *loop* do mesmo estilo, exemplificado adiante:

```
mapRed.preReduce(recordWriter);
while (recordReader.readOneAndAdvance()) {
  mapRed.reduce(recordReader.getCurrentKey(),
                recordReader.getCurrentValue(),
                recordWriter);
}
mapRed.postReduce(recordWriter);
```

- Onde um **recordReader** sabe ler a entrada no formato (**k, v**) e um **recordWriter** (**w**) sabe escrever a saída no formato correto. O que exatamente é **recordReader** e **recordWriter** varia conforme a etapa da execução. Nas "bordas da tarefa" são objetos vindos do *InputFormat* e *OutputFormat* (o **recordReader** de **map** vem do *InputFormat* e o **recordWriter** de **combine** vem do *OutputFormat*), e nas etapas intermediárias são objetos que serão vistos na seção 5.6.
- Os métodos **pre** são chamados antes do *loop* e os métodos **post** são chamadas após o *loop*, e são necessários no caso do usuário querer computar algo que tem uma natureza global, por exemplo uma média, pois de dentro de uma chamada (**map, reduce, combine**) não existe a noção de começo e fim do processamento, apenas conhece-se o elemento (**k, v**) atual.
- O *framework* possui algumas classes já implementadas que lidam com entrada e saída no formato de arquivos de texto UTF-8. Caso o usuário deseje um comportamento diferente do já disponível, ele precisa implementar o algoritmo que lida com os arquivos da forma desejada, implementando os métodos das interfaces descritas adiante, e nos métodos *getInputFormat* e *getOutputFormat* retornar objetos dos tipos que contêm o algoritmo desejado. Tais classes implementam o processamento nas "bordas da tarefa" (arquivo_entrada → **map, combine** → arquivo_saída).
- Nas etapas intermediárias o modelo é rígido e o usuário não tem controle sobre isso. Os dados são uma sequência de registros lógicos no qual o usuário especificou os tipos, que são transmitidos entre as máquinas como arquivos de objetos serializados através do mecanismo padrão da linguagem Java, por meio das classes que serão vistas na seção 5.6.

5.2 Interface InputFormat

```
public interface InputFormat<K, V> {

  File[] getSplits(File in, int numSplits) throws IOException;

  RecordReader<K, V> getRecordReader(File in) throws IOException;

}
```

- O arquivo de entrada precisa ser um arquivo na máquina do *Master*, que quebrará esse arquivo em *splits* para distribui-lo entre os *Workers*. Após o envio do *split*, esse arquivo ficará salvo no disco do *Worker*, e poderá ser usado para uma execução futura sem que seja necessária a transmissão pela rede novamente. O *Master* apenas envia um split a um *Worker* no caso do *Worker* em questão não possuir um arquivo de mesmo nome em seu disco. A flag **-ov** no *Master* faz com que o *Master* sempre envie o arquivo ao *Worker.*
- Método *getSplits* quebra o arquivo de entrada *in* em *numSplits* partes, escrevendo no disco um arquivo temporário para cada parte. Cada elemento do *File[]* retornado é um *handle* para um dos *splits*, que serão enviados aos *Workers* por meio de chamadas RMI.

- Método *getRecordReader* retorna um objeto que sabe ler os dados do arquivo no formato registro[K, V].
- O *framework* possui uma implementação dessa interface com a classe *TextInputFormat*, que assume como entrada um arquivo de texto com *charset UTF-8*, onde cada elemento da entrada será uma linha do texto, resultante da quebra no carácter *newline*.

5.3 Interface RecordReader

```
public interface RecordReader<K, V> extends Closeable {

  K getCurrentKey() throws IOException;

  V getCurrentValue() throws IOException;

  boolean readOneAndAdvance() throws IOException;

}
```

- Método *getCurrentKey* retorna a chave do registro mais recentemente lido.
- Método *getCurrentValue* retorna o valor do registro mais recentemente lido.
- Método *readOneAndAdvance* tenta ler um registro do arquivo e avançar o ponteiro de leitura. O retorno desse método indica sucesso na leitura, ou fim da entrada.
- O *framework* possui uma implementação dessa interface com a classe *LineRecordReader*, que assume como entrada um arquivo de texto com *charset UTF-8*, e tem como noção de registro[K, V] o *offset* em bytes dentro do arquivo como K, e a *string* resultante da quebra do texto no carácter *newline* como V.

5.4 Interface OutputFormat

```
public interface OutputFormat<K, V> {

  RecordWriter<K, V> getRecordWriter(File out) throws IOException;

}
```

- Método *getRecordWriter* retorna um objeto que sabe escrever registro [K, V] no arquivo de saída no formato correto.
- O *framework* possui uma implementação dessa interface com a classe *TextOutputFormat*, que assume como saída um arquivo de texto com *charset UTF-8*, e tem como retorno do método *getRecordWriter* um objeto *LineRecordWriter<K, V>*.

5.5 Interface RecordWriter

```
public interface RecordWriter<K, V> extends Closeable {

  void write(K key, V value) throws IOException;

}
```

- Método *write* escreve (*key*, *value*) na saída.
- O *framework* possui uma implementação dessa interface com a classe *LineRecordWriter*, que assume como saída um arquivo de texto com *charset UTF-8*, aonde cada registro [K, V] será escrito como uma *string* no formato (K.toString() + "\t" + V.toString() + "\n").

5.6 Partições

```
public abstract void reduce(K2 k, Iterable<V2> values, RecordWriter<K3, V3> w)
  throws IOException;

public abstract void combine(K3 k, Iterable<V3> values, RecordWriter<K4, V4> w)
  throws IOException;
```

Dentro de toda chamada **reduce** e **combine**, é garantido que toda ocorrência de um valor de chave **k** presente na tarefa como um todo, estará presente no *Iterable* **values**. Esse processo é particularmente complicado na etapa **reduce**, pois os dados estão espalhados nas diferentes máquinas *Worker*, diferente da etapa **combine,** onde os dados estão concentrados no *Master*. A implementação deste processo é dividido em algumas etapas que serão descritas adiante, introduzindo também a noção de partição, que é um *split* dos dados intermediários que tem como destino um *Worker* especifico, com intenção de facilitar o processo futuro de agrupamento de valores de mesma chave na mesma máquina. A Figura 11 ilustra o processo entre as etapas **map** e **reduce**, onde a saída da etapa **map** foi quebrada em **numWorkers** partes, e cada *Worker* por meio de *reads* remotos, irá ler os pedaços de sua partição espalhados nas diferentes máquinas da rede. A Figura 12 detalha o interior de um *Worker* e as classes importantes.

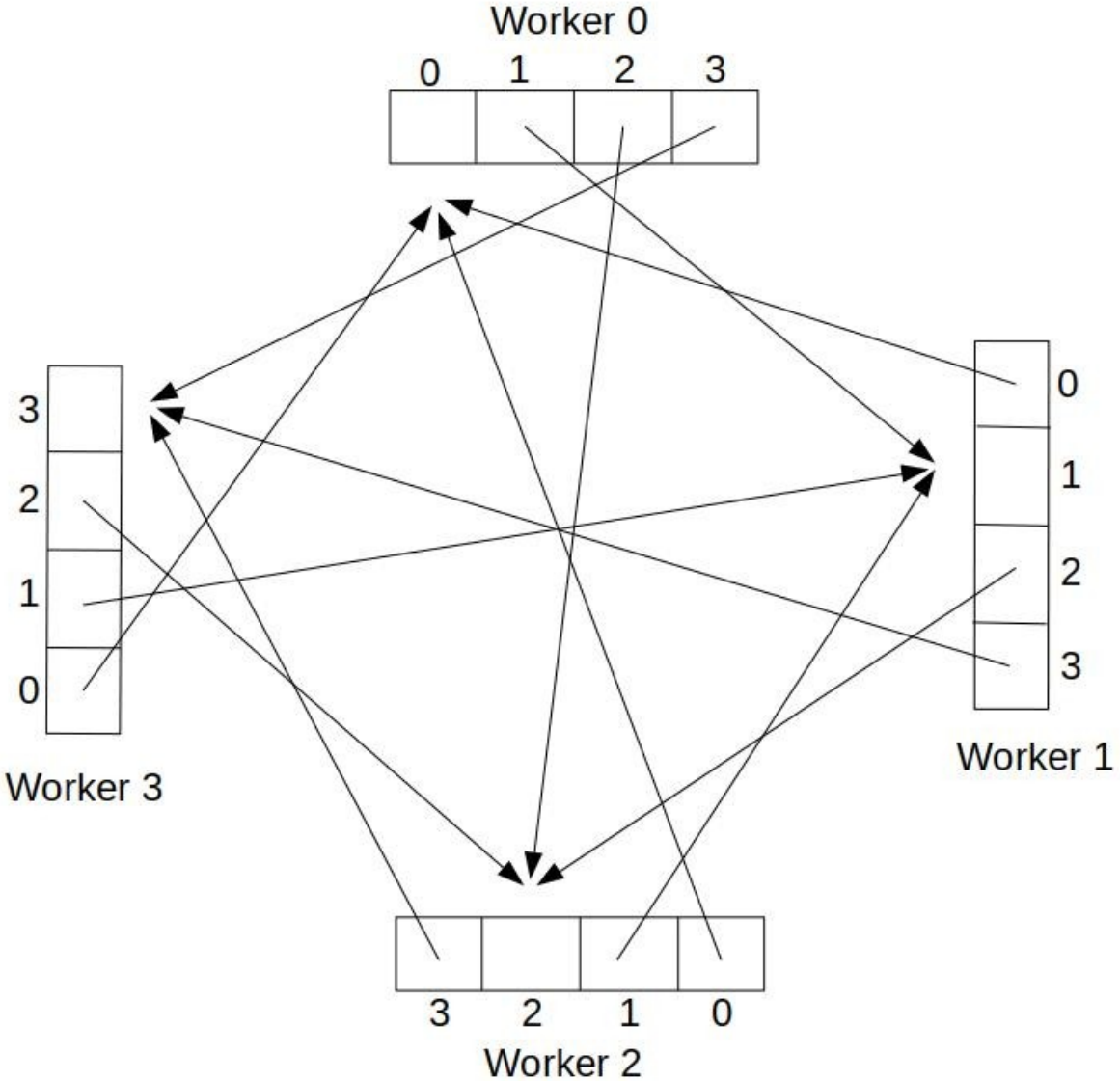

Figura 11: Transferência das partições entre etapas **map** e **reduce**

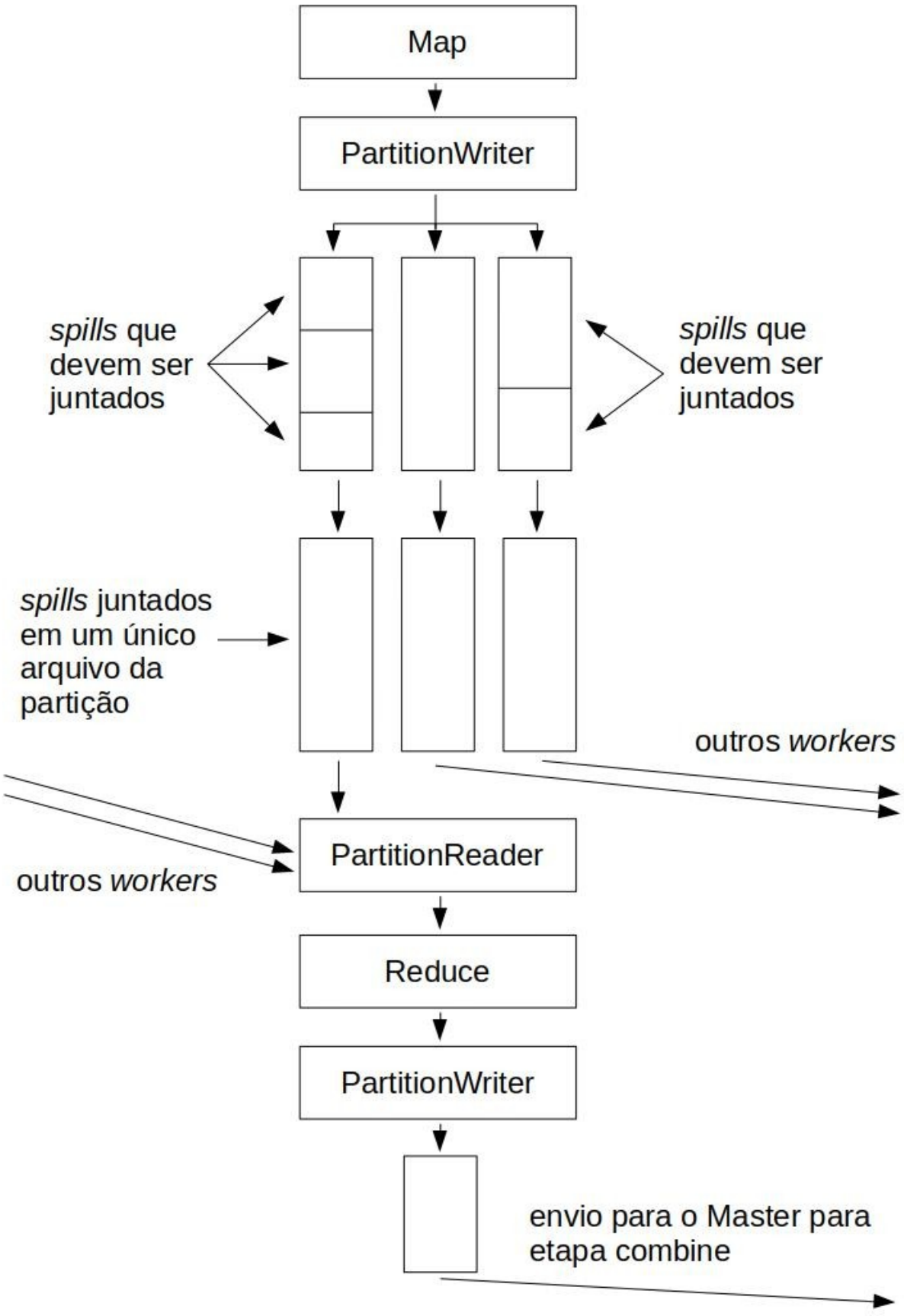

Figura 12: Interior de um *Worker* e classes importantes

PartitionWriter

O **recordWriter** da chamada **map** é um *PartitionWriter,* o qual quebrará a saída da etapa por meio da operação (*hash* da chave **mod** *numWorkers*), a qual divide os dados em **numWorkers** partições mais ou menos do mesmo tamanho, e garante que registros de chave igual tenham como destino a mesma máquina *Worker*, preparando os dados para a etapa **reduce**.

```
int getPartition(K key) {
  return (key.hashCode() & Integer.MAX_VALUE) % parts.length;
}
```

Uma partição é um *array* de **Record<K, V>** em memória, que ao atingir um certo *threshold* será derramado no disco por uma *thread background* como um arquivo de **Record<K, V>** serializados ordenados por **K,** que é chamado de *spill file*. O parâmetro opcional **-bs** no *Master* é o tamanho alocado do *array* e também o valor de *threshold*, que pode ser especificado pelo usuário ou será inicializado com um valor *default* pelo *framework.*

```
ArrayList<Record<K, V>>[] parts;
LinkedList<File>[] spills;
```

parts é o *array* de partições atualmente em memória**,** onde **parts[0]** é o *array* de **Record<K, V>** que corresponde a partição 0. **spills** é um *array* de **LinkedList<File>**, onde **spills[0]** é uma lista com todos os arquivos de *spill* que compõem a partição 0. Ao fim da etapa **map,** uma partição é então o *merge* de todos os seus *spills* em um único arquivo de **Record<K, V>** serializados ordenados, o qual será enviado a um *Worker* específico para a etapa **reduce**. A junção dos arquivos de *spill* é feita com um *k-way merge* de **Record<K, V>** utilizando o comparador de **K,** que é implementado com um *PartitionReader,* classe descrita logo abaixo. O mecanismo de *spill* é trabalhoso e custa tempo, porém é necessário caso a saída seja grande e crescer o *array* **parts** indefinidamente causará erro de memória. Uma saída que pode ser escrita por inteiro no *array* **parts** sem causar erro de memória não precisa de arquivos de *spill*. Como pode ser visto na Figura 12, o **recordWriter** da chamada **reduce** também é um *PartitionWriter*, que tem intenção de escrever os dados no formato correto para a etapa **combine**. No entanto, neste *PartitionWriter* a saída é uma partição única, que será lida pelo *Master* por meio de *reads* remotos em preparação para a etapa **combine**.

PartitionReader

O **recordReader** das chamadas **reduce** e **combine** é um *PartitionReader*, o qual lê um conjunto de arquivos de **Record<K, V>** serializados ordenados por **K**, por meio de um *k-way merge* de arquivos, com intenção de computar o *Iterable* **values** contendo todas ocorrências de mesma chave **k**. O *k-way merge* é implementado com um *PriorityQueue*, do qual retira-se o registro de menor chave enquanto a chave na cabeça da fila não muda. A Figura 13 ilustra o funcionamento do *PartitionReader* durante o *loop* das etapas **reduce** e **combine**, no qual avança-se a entrada e executa-se um *loop* de leitura em toda chamada **readOneAndAdvance**:

```
mapRed.preReduce(recordWriter);
while (recordReader.readOneAndAdvance()) {
  mapRed.reduce(recordReader.getCurrentKey(),
                recordReader.getCurrentValue(),
                recordWriter);
}
mapRed.postReduce(recordWriter);
```

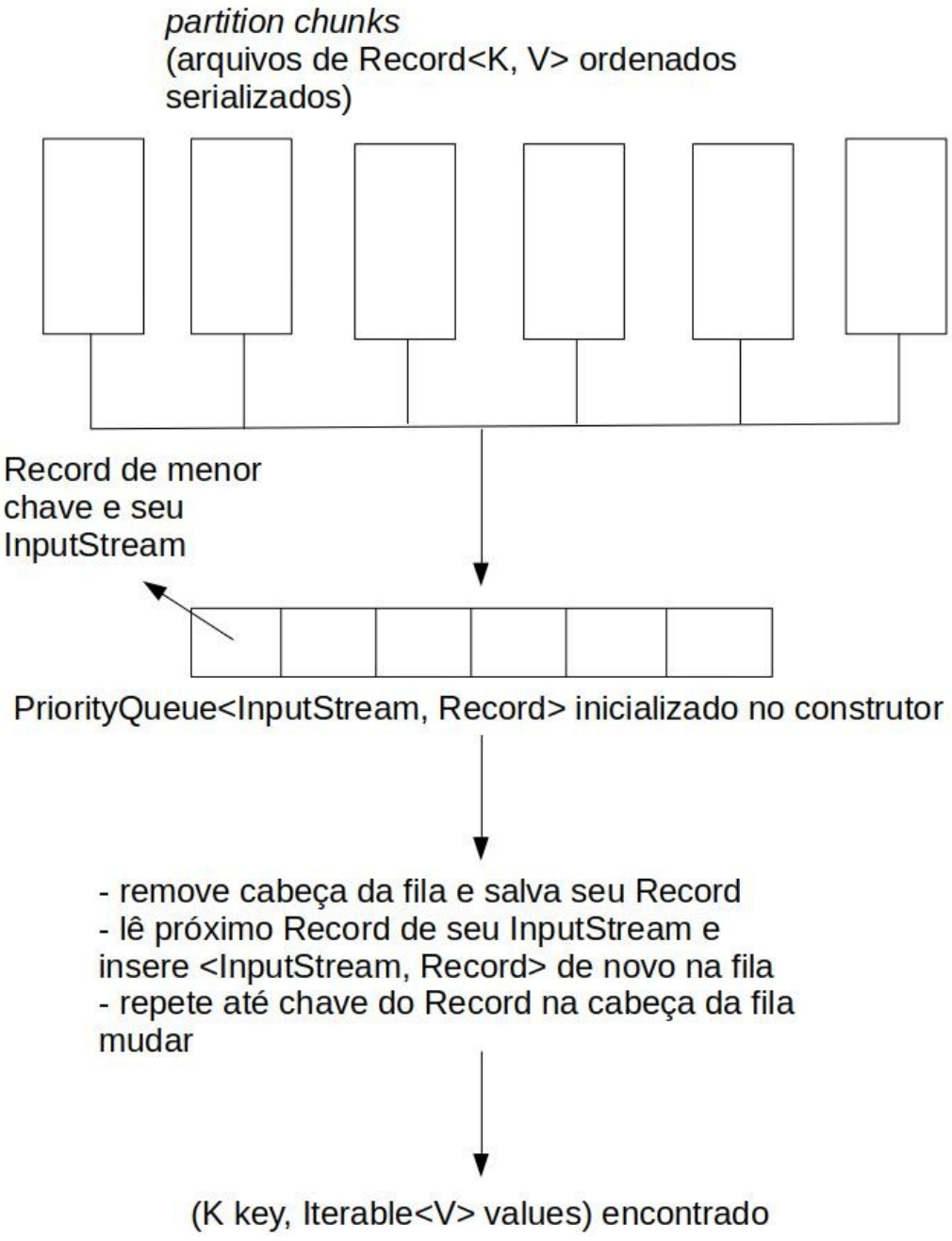

Figura 13: Funcionamento do *PartitionReader*

6 Aplicações exemplo

A seguir alguns casos exemplo de tarefas MapReduce que tem como entrada o arquivo int_base_59.data, o qual possui dados no formato de texto com valores separados por vírgulas agrupados por linha. Figura 14 ilustra um pedaço do arquivo em questão, arquivo o qual também encontra-se no repositório github do projeto.

```
42,30,33,43,96,36,45,55,62,41,55,289,58,70,108,1628,2831
48,22,20,33,40,105,23,26,44,51,39,43,265,53,57,107,2037,2822
42,42,52,27,42,42,82,66,49,56,44,75,49,164,70,51,114,2491,2198
38,81,32,39,58,54,41,57,327,53,63,92,1629,2780
26,32,72,35,34,31,33,35,35,104,43,41,85,4069,1921
74,97,26,35,50,119,32,39,47,63,44,50,199,63,75,135,2028,2793
30,38,85,29,36,58,62,36,57,262,49,71,96,2450,2722
41,55,22,16,33,44,83,23,27,51,61,42,56,288,61,73,99,1491,2888
22,28,40,105,32,32,44,52,39,47,229,53,64,114,2224,2801
32,40,36,24,33,35,77,30,37,46,41,43,49,188,51,50,89,2715,2429
52,39,48,43,60,45,55,63,44,55,60,145,50,45,68,4046,1738
112,42,45,45,32,42,43,100,43,48,54,50,53,57,218,59,2342,2498
```

Figura 14: Formato dos dados do arquivo de entrada da tarefa

Supondo que um usuário precise contar quantas vezes cada *string* separada por vírgula aparecem no arquivo. O arquivo de saída da tarefa MapReduce é exemplificado pela Figura 15, no qual o significado dos valores é: a string “0” apareceu 275 vezes, a string “1” apareceu 61 vezes, a string “10” apareceu 961 vezes, e assim por diante.

```
0     275
1     61
10    961
100   1358
1000  17
1001  29
1002  17
1003  21
1004  21
1005  15
1006  16
1007  23
1008  16
1009  15
```

Figura 15: Formato dos dados do arquivo de saída da tarefa

O código que realiza a computação desejada pelo usuário nesse caso é:

```
public class WordCount extends MapReduce
  <Long, String, String, Integer, String, Integer, String, Integer>
{
  // esta classe é escrita pelo usuário com a computação a ser executada,
  // e vai de entrada no parâmetro -mr do Master, como descrito na
  // seção 5 do documento.

  // utilizando o InputFormat padrão para arquivos de texto já presente no
  // framework. Este InputFormat obrigatóriamente utiliza os tipos
  // <Long, String>, portanto é necessário que os tipos K1 e V1 sejão estes.
  @Override
  public InputFormat<Long, String> getInputFormat() {
    return new TextInputFormat();
  }

  // utilizando o OutputFormat padrão para arquivos de texto já presente no
  // framework. Deseja-se escrever que uma string x apareceu y vezes, portanto
  // utiliza-se os tipos <String, Integer> para os tipos K4 e V4.
  @Override
  public OutputFormat<String, Integer> getOutputFormat() {
    return new TextOutputFormat<String, Integer>();
  }

  // os parâmetros de entrada de cada chamada map são o offset em bytes dentro
  // do arquivo como k, e uma linha do texto como v. Quebra-se v em toda vírgula
  // e escreve na saída que a string s apareceu 1 vez. Utiliza-se os tipos
  // <String, Integer> para os tipos K2 e V2 porque deseja-se dizer que a
  // string x apareceu 1 vez.
  @Override
  public void map(Long k, String v, RecordWriter<String, Integer> w)
    throws  IOException
  {
    for (var s : v.split(",")) {
      w.write(s, 1);
    }
  }

  // os parâmetros de entrada de cada chamada reduce são uma string que
  // apareceu na entrada como k, e um Iterable de todas as suas ocorrências como
  // vs. Soma-se todos os valores em vs e escreve na saída que
  // a string k apareceu sum vezes. Utiliza-se os tipos <String, Integer> para
  // os tipos K3 e V3 porque deseja-se dizer que a string x apareceu y vezes.
  @Override
  public void reduce(String k, Iterable<Integer> vs,
    RecordWriter<String, Integer> w) throws IOException
  {
    int sum = 0;
    for (var v : vs) {
      sum += v;
    }
    w.write(k, sum);
  }
```

```
  // os parâmetros de entrada de cada chamada combine são uma string que
  // apareceu na entrada como k, e um Iterable de todas as suas ocorrências como
  // vs. A etapa combine é um reduce no Master na saída de todos os reduces dos
  // Workers, que neste caso simples apenas escreve a entrada na saída, que terá
  // o efeito de juntar todos os pedaços da saída em um único arquivo. A saída
  // de combine é escrita pelo OutputFormat retornado da função getOutputFormat.
  @Override
  public void combine(String k, Iterable<Integer> vs,
    RecordWriter<String, Integer> w) throws IOException
  {
    for (var v : vs) {
      w.write(k, v);
    }
  }
}
```

Supondo agora que um usuário precise descobrir quais são as 5 *strings* que mais aparecem no arquivo int_base_59.data. Este problema tem várias semelhanças com o exemplo anterior, no entanto as etapas **reduce** e **combine** precisam escrever na saída apenas as 5 *strings* de maior quantidade de ocorrências, em vez de todas. Figura 16 exemplifica o arquivo de saída da tarefa, no qual o significado dos valores é: a *string* “44” apareceu 9319 vezes, a *string* “47” apareceu 9306 vezes, a *string* “45” apareceu 9304 vezes, e assim por diante.

```
44    9319
47    9306
45    9304
46    9289
43    9160
```

Figura 16: Formato dos dados do arquivo de saída da tarefa

O código que realiza a computação desejada pelo usuário nesse caso é:

```
public class WordCountFiveGreater extends MapReduce
  <Long, String, String, Integer, String, Integer, String, Integer>
{
  // Mesma ideia do exemplo anterior
  @Override
  public InputFormat<Long, String> getInputFormat() {
    return new TextInputFormat();
  }

  // Mesma ideia do exemplo anterior
  @Override
  public OutputFormat<String, Integer> getOutputFormat() {
    return new TextOutputFormat<String, Integer>();
  }
```

```
@Override
public void map(Long k, String v, RecordWriter<String, Integer> w)
  throws IOException
{
  for (var s : v.split(",")) {
    w.write(s, 1);
  }
} // Mesma ideia do exemplo anterior

@Override
public void reduce(String k, Iterable<Integer> vs,
  RecordWriter<String, Integer> w) throws IOException
{
  int sum = 0;
  for (var v : vs) {
    sum += v;
  }
  w.write(k, sum);
} // Mesma ideia do exemplo anterior

// o ideal é que a etapa reduce descubra os 5 maiores locais, e a etapa
// combine descubra os 5 maiores globais dentre os 5 maiores locais de todos.
// esse código é muito longo para colocar aqui, portanto simplifica-se a
// solução e a descoberta dos 5 maiores acontece toda na etapa combine.

// estrutura de dados abaixo é uma árvore ordenada pela chave
// (neste caso Integer invertido, o maior vem na frente),
// e usa-se esta estrutura para resolver o problema de
// quais são as 5 strings de maior quantidade de ocorrências, aceitando
// empate (se várias strings aparecem 1000 vezes deseja-se salvar todas).
// Esta árvore mantém sempre os 5 maiores valores de ocorrências e listas das
// strings com as respectivas contagens de ocorrências. Exemplo:
// - a entrada de chave 10 é uma lista das strings que aparecem 10 vezes
// - a entrada de chave 55 é uma lista das strings que aparecem 55 vezes
// A etapa combine vai alterando as entradas da árvore, e ao fim de sua
// execução tem-se aqui a resposta do problema.

TreeMap<Integer, LinkedList<String>> map
  = new TreeMap<>(Comparator.reverseOrder());

// escreve na saída ao fim da execução da etapa combine, o conteúdo
// do TreeMap map. No caso do exemplo do arquivo int_base_59.data, esta
// chamada irá escrever o conteúdo da Figura 16.
// en.getKey() é uma quantidade de ocorrências e en.getValue() é uma lista com
// as strings que tem a respectiva quantidade. Portanto o loop escreve todas
// strings de en.getValue() que tiveram quantidade de ocorrências
// en.getKey(). No caso do arquivo int_base_59.data não existe empate, mas
// supondo que uma outra string também tenha contagem 9319, ela estaria logo
// antes ou logo após o “44”, e a saída teria 6 linhas com 6 strings e suas
// quantidades de ocorrências (o algoritmo admite empates).

@Override
public void postCombine(RecordWriter<String, Integer> w) throws IOException {
  for (var en : map.entrySet()) {
    for (var s : en.getValue()) {
      w.write(s, en.getKey());
}}}
```

```
  // uma chamada combine decide se a string k com quantidade de ocorrências cnt
  // deve entrar na árvore das 5 maiores quantidades de ocorrências.
  // A operação cnt = ((LinkedList<Integer>) vs).getFirst() neste caso é válida
  // pois tem-se certeza que vs é sempre uma lista com 1 item. A lógica do
  // algoritmo é a seguinte:
  // - se a árvore tem menos que 5 entradas, k entra na árvore
  // com o novo nó (cnt, list(k)).
  // - se a árvore já tem uma entrada de chave cnt, k entra na lista das
  // strings do nó cnt.
  // - se cnt é maior do que a entrada de menor valor da árvore, k entra na
  // árvore com o novo nó (cnt, list(k)), e retira-se da árvore a entrada de
  // menor valor.
  // A execução da função combine em todos os elementos que saíram da etapa
  // reduce faz com que ao fim, TreeMap map tenha a resposta desejada do
  // problema.

  @Override
  public void combine(String k, Iterable<Integer> vs,
    RecordWriter<String, Integer> w) throws IOException
  {
    int cnt = ((LinkedList<Integer>) vs).getFirst();
    if (map.size() < 5) {
      var l = new LinkedList<String>();
      l.add(k);
      map.put(cnt, l);
    }
    else if (map.containsKey(cnt)) {
      map.get(cnt).add(k);
    }
    else if (cnt > map.lastKey()) {
      var l = new LinkedList<String>();
      l.add(k);
      map.put(cnt, l);
      map.remove(map.lastKey());
    }
  }
}
```

7 Conclusão

O modelo MapReduce foi um sucesso no mundo utilizado na solução de variados problemas, sucesso o qual foi consequência de algumas importantes características. Primeira e principal, o modelo é fácil de se usar, até por programadores sem domínio de sistemas distribuídos, pelo fato de implementar internamente e esconder do usuário detalhes complexos da computação distribuída. Isso permite ao usuário lidar apenas com uma interface relativamente simples. Segunda, uma grande variedade de problemas podem ser expressos como computações MapReduce sem grandes dificuldades. Internamente no Google, MapReduce foi utilizado na solução de problemas como: geração de dados para o serviço de busca, para mineração de dados e aprendizado de máquina, para ordenação de arquivos gigantes, e muitos outros problemas [1]. Terceira, o modelo MapReduce escala bem para execução em *clusters* de centenas ou milhares de máquinas, pois faz um uso eficiente e bem distribuído dos recursos limitados das máquinas, tornando-o adequado para a utilização em problemas de processamento *big data*.

Referências bibliográficas

[1] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In EuroSys 2007, pages 59–72, 2007.

[3] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In SIGMOD ’07, pages 1029–1040. ACM, 2007.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica. Spark: Cluster Computing with Working Sets. HotCloud 2010. June 2010.

[5] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, M. Zaharia. Scaling Spark in the Real World: Performance and Usability. Proc. VLDB Endow., 8, 1840-1843, 2015.

[6] T. White, Hadoop: The Definitive Guide. Beijing: O'Reilly. (2015)

[7] ASM. https://asm.ow2.io acesso 19/05/2021.

[8] Oracle, https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/ acesso 19/05/2021.