PUCRS online

APRESENTAÇÃO DE APOIO

GERÊNCIA DE INFRAESTRUTURA PARA BIG DATA

Tiago Coelho Ferreto – Aula 05

Pós-Graduação em

Ciência de Dados e Inteligência Artificial

Ementa da disciplina

Introdução à arquitetura para Big Data Analytics. Visão geral sobre Infraestrutura de armazenamento de dados para Big Data. Visão geral sobre Infraestrutura de computação e de rede para Big Data. Tópicos sobre virtualização e computação em nuvem. Plataformas de Big Data na nuvem: HDFS, Hadoop e MapReduce. Estudos de caso com Spark.

Professores

MARCOS TAKESHI
Professor Convidado

Especialista em Big Data na Semantix, que atua em diversos projetos de empresas do setor financeiro, telecom, varejo e saúde. Realiza análises de arquiteturas, infraestruturas, ambientes, sistemas e ferramentas big data, visando o correto funcionamento e performance. Formado em engenharia eletrônica pela Escola de Engenharia Mauá, pós-graduado em Administração de Empresas pela FGV-SP, MBA em Big Data na FIAP, e empreendedorismo no Babson College.

TIAGO COELHO FERRETO
Professor PUCRS

É professor adjunto da Pontifícia Universidade Católica do Rio Grande do Sul. Possui Doutorado em Ciência da Computação pela PUCRS (2010) com Doutorado sanduíche na Technische Universität Berlin, Alemanha (2007-2008). Tem experiência na área de Ciência da Computação, com ênfase em Redes de Computadores, atuando principalmente nos seguintes temas: computação em nuvem, grades computacionais, virtualização, processamento de alto desempenho e gerência de infraestrutura de TI.

Encontros e resumo da disciplina

AULA 1

Para ser um profissional de Data Science é necessário ter paciência e construir um bom Network.

Empresas tem grande interesse em processar os dados e deles extrair informação com a finalidade de monetizar.

É bom estar no meio de pessoas que saibam mais do que você, sempre você tem que estar no meio de pessoas melhores.

MARCOS TAKESHI
Professor Convidado

AULA 2

O Spark possibilita a obtenção de resultados imediatos.

É importante você saber e conseguir atuar em mais de uma frente.

Certificações podem mostrar que você tem conhecimento do assunto.

MARCOS TAKESHI
Professor Convidado

AULA 3

Nos últimos anos a gente tem, a cada ano, um novo software auxiliando no processamento de grandes volumes de dados.

Além de armazenar e processar, eu tenho que conseguir extrair valor.

O Hadoop como a principal ferramenta para trabalhar com grandes volumes de dados.

TIAGO COELHO FERRETO
Professor PUCRS

AULA 4

A redundância garante a persistência da informação.

O HDFS é a principal fonte de dados de entrada e saída do Hadoop.

Como utilizar as aplicações Sqoop e Flume.

TIAGO COELHO FERRETO
Professor PUCRS

AULA 5

MapReduce uma solução de escalonamento e capacidade de processamento.

Hadoop Streaming como implementação de funções Map e Reduce em linguagens diferentes de Java.

O Pig como linguagem alternativa para programar MapReduce.

TIAGO COELHO FERRETO
Professor PUCRS

AULA 6

O Hive trabalha com a linguagem SQL com interações através de linhas de comando em formato shell.

O Spark tem como benefícios uma melhor performance, extensibilidade e melhor suporte para outros cenários.

O componente principal do Spark é o RDD (Resilient Distributed Dataset).

TIAGO COELHO FERRETO
Professor PUCRS

Ciências de Dados e Inteligência Artificial

Gerência de Infraestrutura para Big Data

MapReduce

Prof. Tiago Ferreto

tiago.ferreto@pucrs.br

PUCRS online UOL edtech_

Introdução ao MapReduce

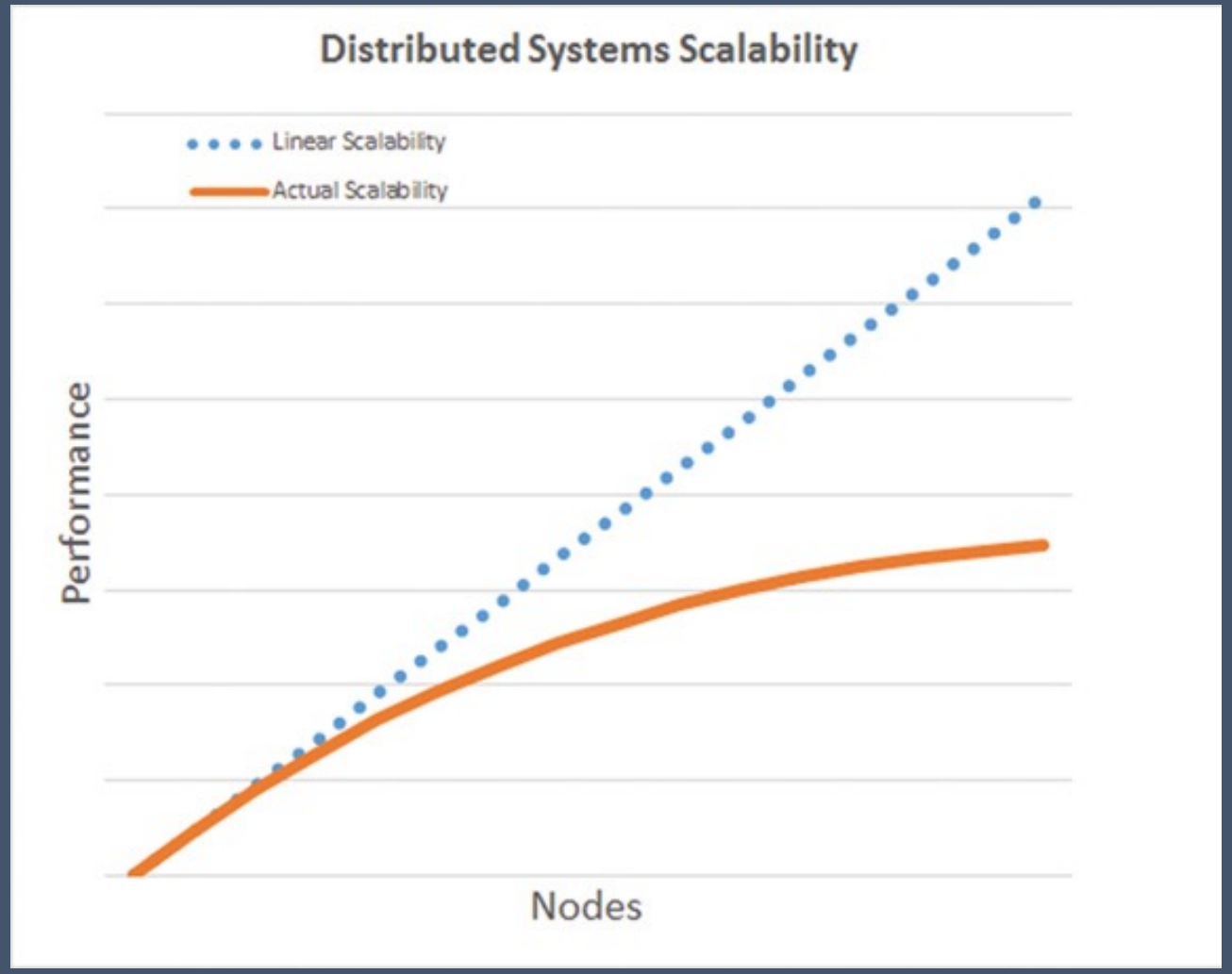
- White paper “ **MapReduce: Simplified Data Processing on Large Clusters** ”, lançado em dezembro de 2004 pelo Google
  - Descrição de alto nível da abordagem do Google para processamento, especificamente indexação e classificação, de grandes volumes de dados de texto para processamento do mecanismo de busca

- Influência no **projeto Nutch** (Yahoo!)
  - Os criadores do projeto Nutch (incluindo **Doug Cutting**) incorporaram os princípios descritos nos documentos Google MapReduce e Google File System no projeto agora conhecido como **Hadoop**

Motivação

- **Limitações na abordagem de escalonamento** para aumentar a capacidade de processamento
- Antes do MapReduce (2004), haviam vários frameworks de programação para sistemas distribuídos - Message Passing Interface (MPI), Parallel Virtual Machine (PVM), HTCondor e outros
- Mas eles tinham várias limitações:
  - **Complexidade na programação**: necessidade de lidar explicitamente com o estado e a sincronização entre processos distribuídos, incluindo dependências temporais
  - **Falhas parciais (difíceis de recuperar):** sincronização e troca de dados entre processos em um sistema distribuído tornou o tratamento de falhas parciais muito mais desafiador
  - **Gargalos na transferência de dados para o processador**: a maioria dos sistemas distribuídos obtém dados de armazenamento compartilhado ou remoto
  - **Escalabilidade limitada**: largura de banda finita entre processos limita como os sistemas distribuídos podem escalar

Problemas de escalabilidade em sistemas distribuídos



Metas de projeto para o MapReduce

- **Paralelização e distribuição automáticas**: o modelo de programação deve facilitar a paralelização e distribuição dos cálculos
- **Tolerância a falhas**: o sistema deve ser capaz de lidar com falhas parciais. Se um nó ou processo falhar, sua carga de trabalho deve ser assumida por outros componentes em funcionamento no sistema
- **Escalonamento de entrada/saída**: o escalonamento e alocação de tarefas visam limitar a quantidade de largura de banda da rede usada. As tarefas são agendadas dinamicamente nos nós disponíveis para que os trabalhadores mais rápidos processem mais tarefas
- **Status e monitoramento**: o status de cada componente e suas tarefas em execução, incluindo progresso e contadores, são relatados a um processo mestre. Isso facilita o diagnóstico de problemas, a otimização de tarefas, e o planejamento da execução

MapReduce - Pares chave-valor e registros

- Os registros de entrada, saída e intermediários no MapReduce são representados na forma de pares de chave-valor
  - Os pares de chave-valor são comumente usados na programação para definir uma unidade de dados
- **A chave** é geralmente um identificador (por exemplo, nome do atributo)
  - Em alguns sistemas, a chave precisa ser exclusiva em relação a outras chaves no mesmo sistema (por exemplo, NoSQL), mas isso **NÃO** É **REQUERIDO no MapReduce**
- **Valor** são os dados que correspondem à chave
  - Pode ser um valor escalar simples, como um inteiro, ou um objeto complexo, como uma lista de outros objetos

Key	Value
City	Chicago
Temperatures	[35,38,27,16]

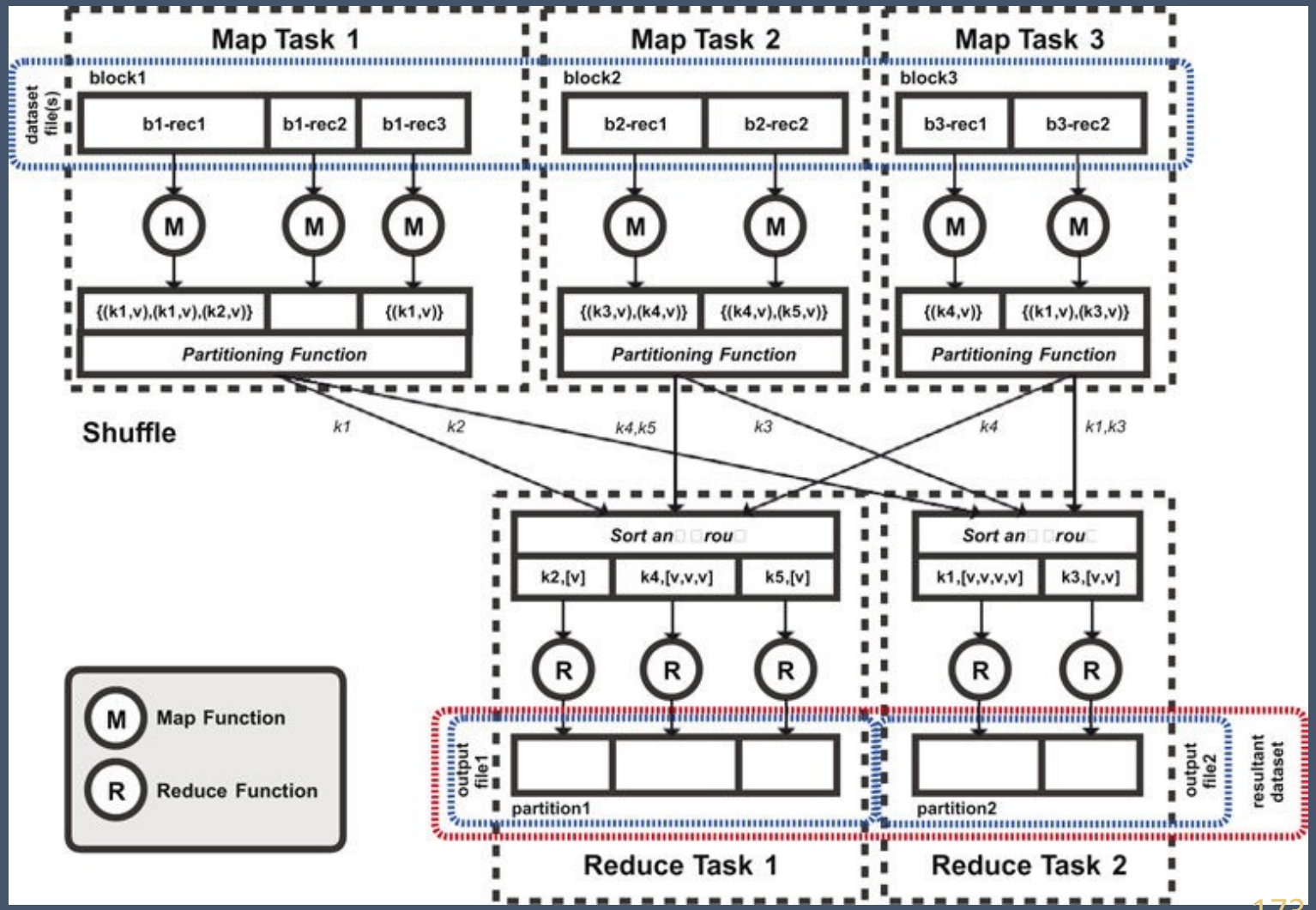
MapReduce - Pares chave-valor e registros

- Os pares de chave-valor são implementados em muitas linguagens de programação
  - Exemplo: Python usa dicionários. Ruby usa hashes
- **Os pares de chave-valor são a unidade de dados atômico usada para processamento na programação MapReduce**

- Problemas complexos são frequentemente decompostos no Hadoop em uma série de operações em pares de chave-valor

Modelo de programação MapReduce

- Inspirado nas primitivas map e reduce do Lisp e de outras linguagens de programação funcionais
- MR inclui:
  - Duas fases de processamento implementadas pelo desenvolvedor: **fase de mapeamento** e **fase de redução**
  - Fase implementada pelo framework: **Shuffle-and-sort**

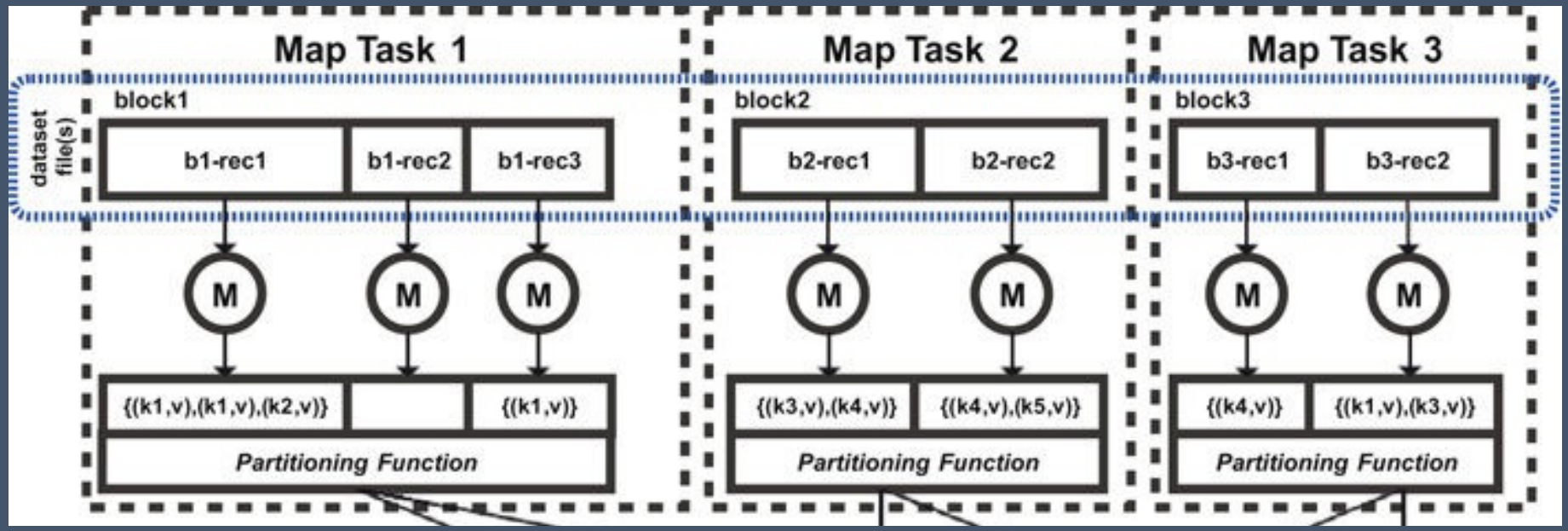


Fase Map

- Fase inicial para processar um conjunto de dados de entrada
- **O formato de entrada** (Input format) e as **funções do leitor de** registro (record reader) derivam registros na forma de pares chave-valor para os dados de entrada
- A fase Map aplica uma função ou funções a cada par de chave-valor em uma **parte do conjunto de dados** (bloco no sistema de arquivos HDFS)
  - n blocos de dados no conjunto de dados de entrada → n tarefas Map (também chamadas de mapeadores - *mappers*)
- NENHUM ESTADO É COMPARTILHADO entre processos
  - Cada **tarefa Map itera por meio de sua parte** do conjunto de dados EM PARALELO com as outras tarefas Map
  - Cada **registro (par de chave-valor) é processado** UMA VEZ E APENAS UMA VEZ (com exceção apenas para falha de tarefa ou execução especulativa)

Fase Map - Exemplo

- **Três tarefas Map** operando contra **três blocos** (bloco1, bloco2 e bloco3)
- A tarefa Map chama sua função Map (M) uma vez para cada registro (par de chave-valor)
- A função Map recebe um par de chave-valor e produz zero ou mais pares de chave-valor
  - Os resultados são considerados DADOS INTERMEDIÁRIOS (podem ser processados posteriormente pela fase Reduce)



Fase Map

- Pseudo-código
  - `map (in_key, in_value) → list (intermediate_key, intermediate_value)`

- Exemplos
  - Filtrando mensagens de log (apenas mensagens de ERROR)
    - `let map (k, v) = if (ERROR in v) then emit (k, v)`
  - Manipular valores (converter para minúsculas)
    - `let map (k, v) = emit (k, v.toLower( ))`

Fase Map

- **Não** podem haver **dependências** entre as tarefas Map
  - Qualquer função Map é válida se a função puder ser executada em um registro contido em uma parte do conjunto de dados, isoladamente de outras tarefas Map que estão processando outras partes do conjunto de dados
- A tarefa Map coleta listas de pares de chave-valor intermediários emitidos por cada função Map em uma única lista agrupada pela chave intermediária
- A lista combinada de valores intermediários agrupados por suas chaves intermediárias é então passada para uma **função de particionamento**

Função de particionamento (ou particionador)

- Objetivo: garantir que cada chave e sua lista de valores sejam passados para uma e apenas uma tarefa Reducer (redutor)
  - Implementação mais comum: **particionador hash**
    - Cria um hash (ou assinatura única) para a chave e divide o espaço da chave com hash em n partições (onde n é o número de redutores)
- Particionadores personalizados também podem ser implementados
  - Exemplo: particionador que divide os dados por mês para processar dados de um ano
- Particionador (chave) → Redutor
  - A função de particionamento é chamada para cada chave com uma saída representando o Redutor de destino para a chave, normalmente um número entre 0 e n - 1 (n = número de Redutores)

Shuffle and Sort

- A saída de cada tarefa Map é enviada para uma tarefa Reduce destino (definida pela função de particionamento)

- Os dados são transferidos fisicamente entre os nós (**requer uso da rede**)

- As chaves e seus valores são **agrupados** e **apresentados de forma ordenada em relação a chave** para o Redutor alvo (SORT)
  - Exemplo: se a chave for um valor de texto, as chaves serão apresentadas ao Redutor em ordem alfabética crescente

Fase Reduce

- Condição inicial
  - Todos os mappers concluíram E
  - A fase de shuffle-and-sort transferiu todas as chaves intermediárias e suas listas de valores intermediários para seu redutor alvo
- Redutor (ou tarefa de Redução) executa uma função reduce para cada chave intermediária e sua lista de valores intermediários associados
- A saída de cada função reduce é **zero ou mais pares de chave-valor** considerados parte da SAÍDA FINAL
  - A saída pode ser a entrada para outra fase Map em um fluxo de trabalho computacional complexo de vários estágios

Fase Reduce

- Representação de pseudo-código:
  - `reduce (key, list (intermediate_value)) → key, out_value`
- As funções reduce são frequentemente **funções de agrupamento**, como: somas, contagens e médias
- Exemplo: Redutor de Soma

```
let reduce (k, list <v>) =
        sum = 0
        for int i in list <v> :
          sum + = i
        emit (k, sum)
```

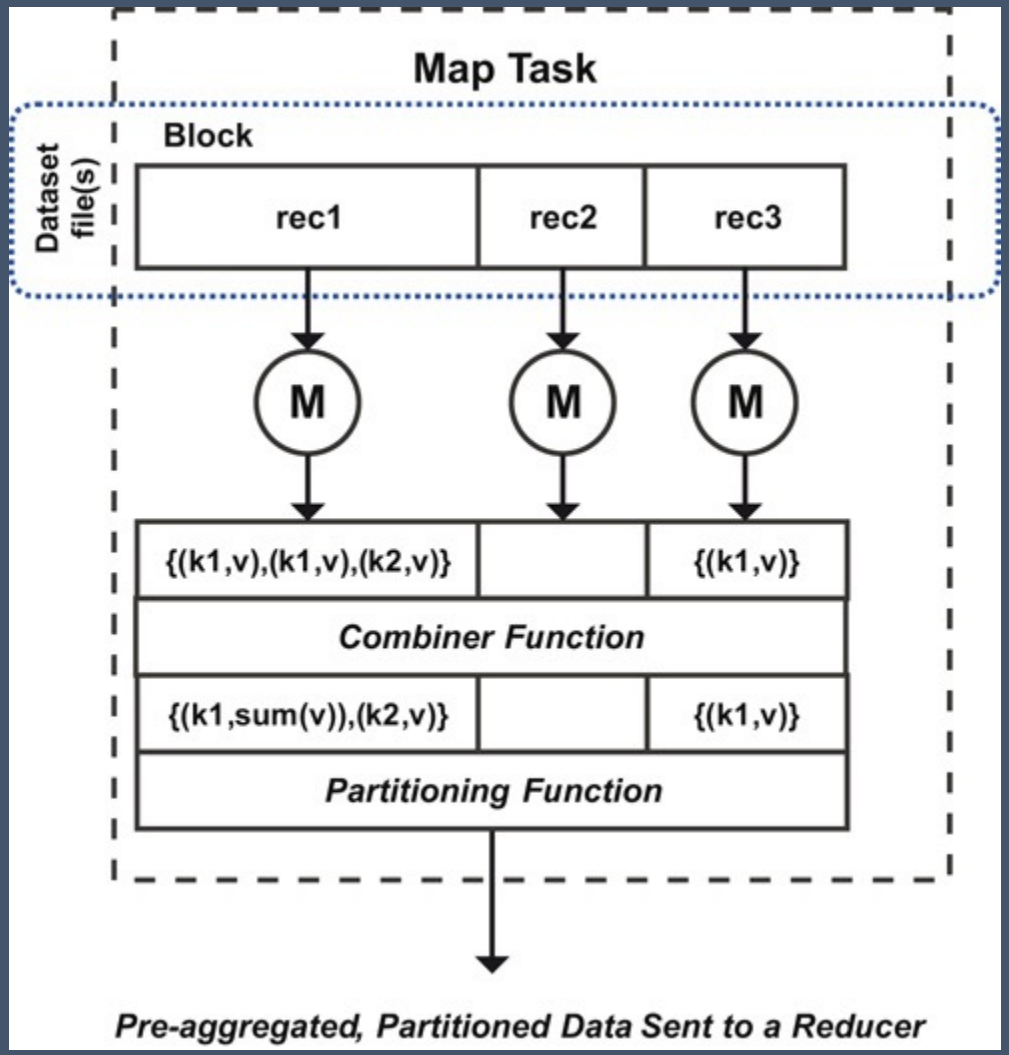
Tolerância a falhas

- MapReduce tolera falhas de nó
- Quando uma tarefa Map falha, ela é automaticamente reprogramada pelo processo mestre para outro nó, de preferência um nó que possui uma cópia do(s) mesmo(s) bloco(s), mantendo a localidade dos dados (caso contrário, acessará o bloco através da rede)
- Uma tarefa pode falhar e ser reprogramada **quatro vezes** antes que o trabalho seja considerado como tendo falhado
- Se uma tarefa Reduce falhar, ela também pode ser reprogramada e seus dados de entrada reabastecidos
  - Dados intermediários são mantidos durante todo o tempo de execução da aplicação

Função Combiner (Combinador)

- Se as operações de **redução** forem **comutativas e associativas** (por exemplo, somas e contagens), as operações podem ser realizadas após a tarefa Map ser concluída no nó que a executa (antes da fase de Shuffle-and-Sort)
  - Utilização de uma função Combiner
- Operações não comutativas e associativas (por exemplo, médias) não podem ser implementadas como um combinador
  - Avg (LIST) ≠ Avg (Avg (SUBLIST), Avg (SUBLIST))
  - Exemplo: Avg (2,2,3,3,3) → 2,6, enquanto Avg (Avg (2,2), Avg (3,3,3)) = Avg (2,3) → 2,5
- Combinador diminui
  - Quantidade de dados transferidos na fase Shuffle
  - Carga computacional na fase de redução
- A função do combinador é frequentemente igual à função reduce, mas é executada no mesmo nó da tarefa Map

Função Combiner



Assimetria e execução especulativa

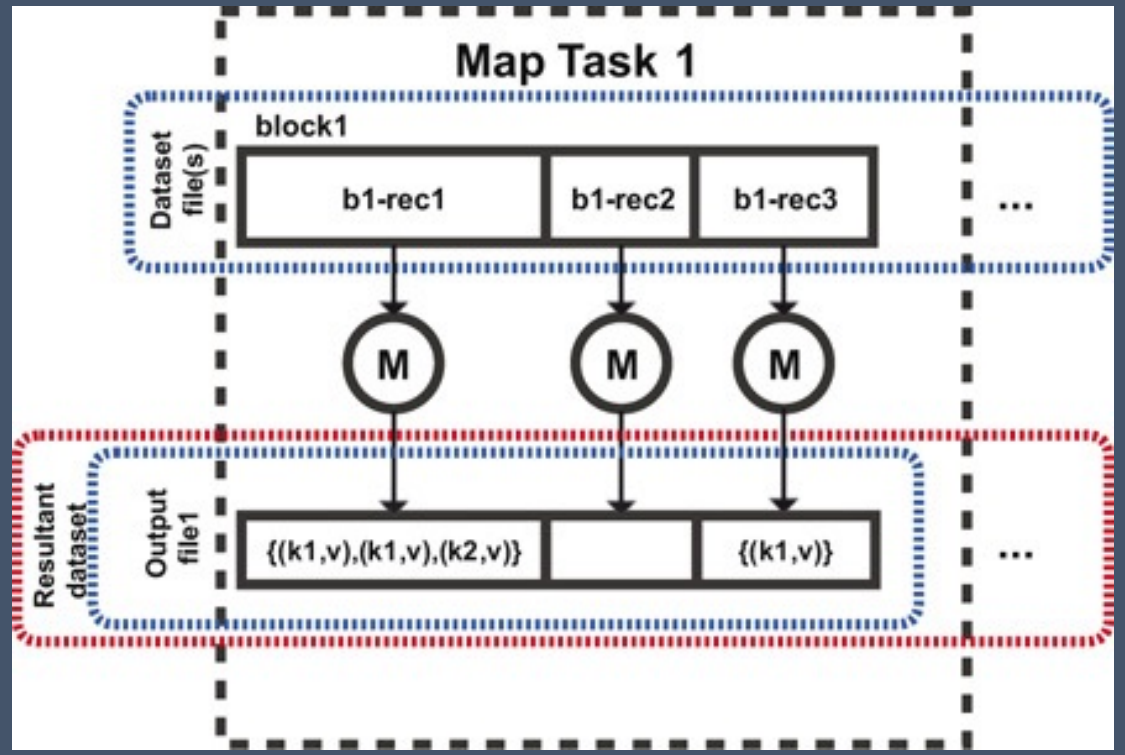
- As fases Map e Reduce são assimétricas
  - Um mapper pode fazer mais processamento do que outro mapper no mesmo conjunto de dados
  - Exemplo: filtragem de weblog para um IP específico → alguns blocos podem conter mais referências ao IP do que outros

- Alguns mapeadores podem executar mais devagar do que outros
  - A fase de reduce começa somente após a fase de mapa ser concluída → **problema de desempenho**: é necessário esperar mapeadores mais lentos!

- **Execução especulativa** (regida pelo ResourceManager e ApplicationMaster)
  - Procura por diferenças toleráveis de andamento entre as tarefas
  - Se uma tarefa ficar fora dessa tolerância (está demorando muito para ser concluída), uma tarefa duplicada é criada para processar os mesmos dados
  - Os resultados da primeira tarefa a ser concluída são usados e a outra tarefa é eliminada (e a saída descartada)
  - Evita que um nó lento, sobrecarregado ou instável se torne um gargalo

Aplicações MapReduce – Map-Only

- Aplicação MapReduce - Map-only → uma aplicação MR com ZERO tarefas Reduce
- Exemplos de uso
  - Rotinas ETL onde os dados não se destinam a ser resumidos, agregados ou reduzidos
  - Trabalhos de conversão de formato de arquivo
  - Trabalhos de processamento de imagem
- Sem função de particionamento → a saída do Mapper é a saída final
- Fornece paralelização massiva, evitando a operação Shuffle-and-Sort (custosa)



Implementação do MapReduce no Hadoop

- Duas implementações
  - Mapreduce cluster framework (MR1 ou MapReduce versão 1) - Hadoop 1
  - YARN (MR2) - Hadoop 2

- MR1 usa os seguintes daemons
  - JobTracker (em vez do ResourceManager no YARN)
  - TaskTracker (em vez do NodeManager no YARN)
- Desvantagens do MR1
  - Não funciona com programas não-MapReduce (por exemplo, Spark)
  - Escalabilidade limitada
  - Uso ineficiente da capacidade de processamento (especialmente em relação a recursos heterogêneos)

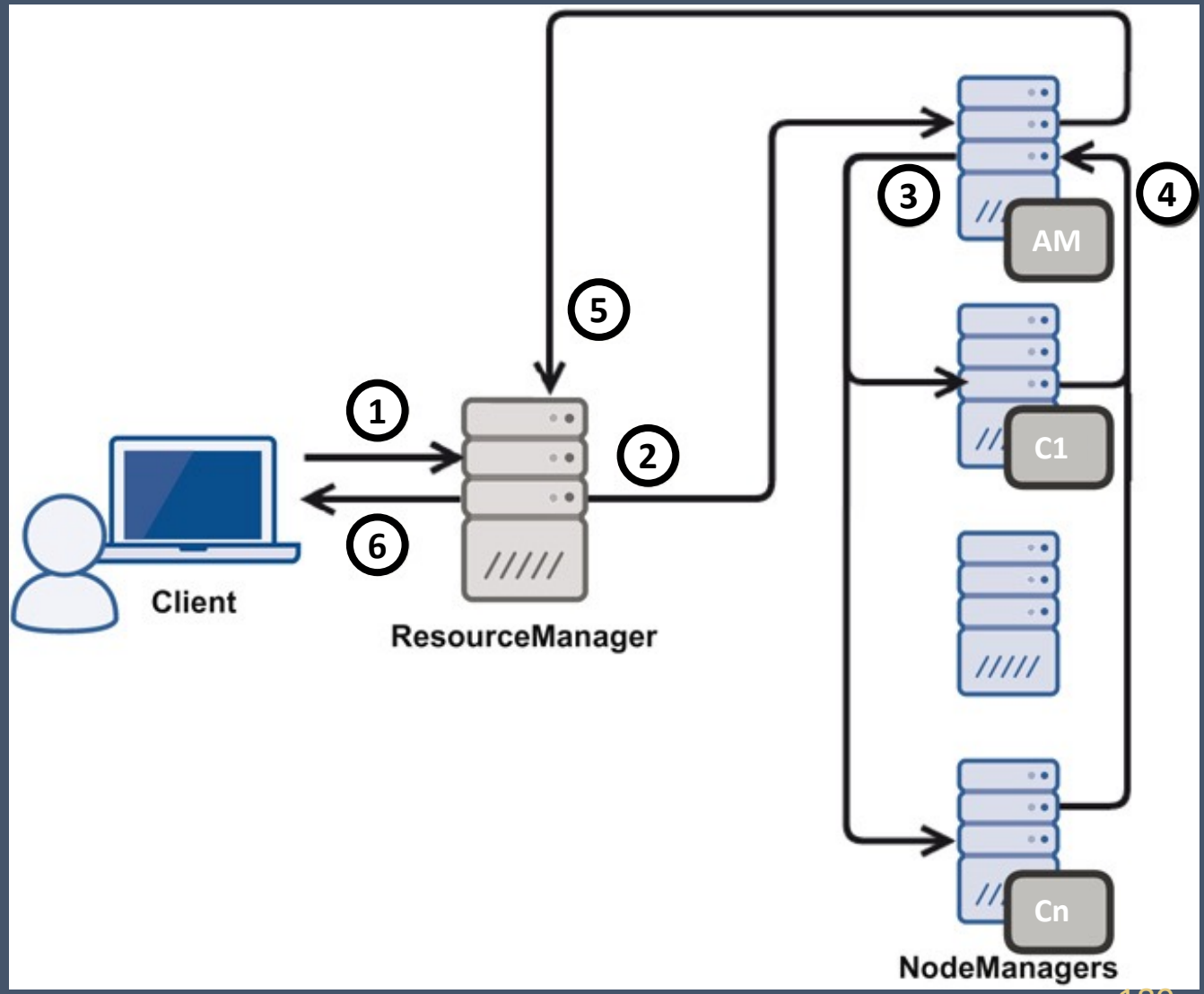
Execução e aplicação no YARN

- **O YARN escalona e orquestra** aplicações e tarefas no Hadoop
  - Aplica o conceito de localidade de dados - as tarefas são agendadas no nó onde os dados residem
- **A carga de trabalho do aplicativo é distribuída** entre NodeManagers
  - Os NodeManagers são responsáveis pela execução de tarefas
- ResourceManager (mestre do YARN) é responsável por
  - **Atribuição de um ApplicationMaster** (processo de delegação para gerenciar a execução e o status de uma aplicação)
  - **Acompanhar os recursos disponíveis** nos NodeManagers, como núcleos de CPU e memória
- Recursos de computação e memória são apresentados aos aplicativos em unidades de processamento chamadas de **containers**

- **ApplicationMaster determina os requisitos de container** para o aplicativo e negocia esses recursos com o ResourceManager

YARN – submissão, escalonamento e execução

1. O cliente envia a aplicação ao ResourceManager
2. ResourceManager aloca um ApplicationMaster em um NodeManager com capacidade suficiente
3. ApplicationMaster negocia containers de tarefa com o ResourceManager para serem executados em NodeManagers (inclui o NodeManager atual) e despacha o processamento para NodeManagers que hospedam containers de tarefa para a aplicação
4. Os NodeManagers relatam o status e o progresso ao ApplicationMaster
5. ApplicationMaster relata progresso e status para Resourcemanager
6. ResourceManager relata o progresso, status e resultados da aplicação para o cliente



Fase Map

- Tarefas de Map e Reduce são agendadas para execução em containers em execução em NodeManagers

- As tarefas Map são agendadas de acordo com **InputSplits**
  - InputSplits são limitados pelo tamanho do bloco HDFS para os dados de entrada (pode ser considerado equivalente a blocos HDFS)

- O ApplicationMaster (AM) tenta escalonar as tarefas Map nos mesmos nós que contêm os blocos que compõem o conjunto de dados de entrada para a aplicação

- O ApplicationMaster monitora o progresso das tarefas Map

Fase Shuffle-and-Sort

- Após a conclusão de uma porcentagem de tarefas Map, as tarefas Reduce são agendadas (parâmetro *mapreduce.job.reduce.slowstart.completed.maps* em mapred-site.xml)
  - Identificação de NodeManagers com recursos suficientes para instanciar containers para executar as tarefas Reduce
  - A fase Reduce não requer localidade dos dados (os dados intermediários são transferidos para o nó)
  - O número de tarefas Reduce é especificado pelo desenvolvedor

- Depois de concluir todas as tarefas Map e criar containers de Redutor, os dados intermediários das tarefas Map (chaves e listas de valores) são enviados para o Reducer apropriado (com base na função de particionamento)
  - **A** transferência de **dados intermediária** é do disco local no nó da tarefa Map para o disco local no nó da tarefa Reduce (**não usa HDFS!**)

- Chaves e listas de valores são mescladas em uma lista por Reducer
  - Chaves armazenadas em ordem de classificação de chave de acordo com o tipo de dados da chave

Fase Reduce

- O Reduce só começa após a conclusão da fase Shuffle-and-Sort
- O Reduce é executado para cada chave intermediária até que todas as chaves tenham sido processadas
- Normalmente, um trabalho MapReduce gravará dados em um diretório de destino no HDFS
  - **Cada tarefa Reduce grava seu próprio arquivo de saída** (part-r-nnnnn, onde nnnnn é o identificador do Redutor)
  - Os dados podem ser usados como entrada para processamento subsequente (outra aplicação MapReduce)
- Depois que todas as tarefas Reduce forem concluídas e sua saída tiver sido gravada no HDFS, a aplicação estará completa
  - ApplicationMaster informa o ResourceManager, e os containers e recursos usados para a aplicação são liberados

Java MAPREDUCE API

Java MapReduce API – Conceitos Básicos

- A serialização está amplamente presente na API Java MapReduce
  - Conversão de estruturas de dados em fluxos de bytes e vice-versa (desserialização)
- Tipos de dados Hadoop
  - Chaves e valores são objetos serializáveis
    - Utilização de tipos de objetos de dados integrados (Box Classes) em vez de tipos primitivos (int, long, char)
    - Requer o uso de métodos específicos para acessar / modificar dados
  - Todos os objetos serializáveis usam a interface Writable
    - Define métodos de acesso e modificadores
    - Exemplo: métodos readFields e write
  - A classificação é fornecida por uma interface WritableComparable
    - Fornece métodos compareTo, equals e hashCode

Hadoop Box Class	Java Primitive
BooleanWritable	boolean
ByteWritable	byte
IntWritable	int
FloatWritable	float
LongWritable	long
DoubleWritable	double
NullWritable	null
Text	String

Java MapReduce API – Conceitos Básicos

- **InputFormats**
  - Especifica como os dados (chaves e valores) são extraídos de um arquivo
  - Fornece uma *fábrica* para objetos RecordReader →usados para extrair dados de um InputSplit
  - Exemplos de InputFormats
    - **TextInputFormat** : InputFormat para arquivos simples. Os arquivos são divididos em linhas. As chaves são a posição no arquivo e os valores são a linha de texto
    - **KeyValueTextInputFormat** : semelhante a TextInputFormat, mas com cada linha dividida em uma chave e valor por um separador
    - **SequenceFileInputFormat** : InputFile para SequenceFiles (formato Hadoop especial)
    - **NLineInputFormat** : semelhante a TextInputFormat e especifica quantas linhas devem ir para cada tarefa Map
    - **DBInputFormat** : InputFormat para ler dados de uma fonte de dados JDBC
    - **FixedLengthInputFormat** : InputFormat para ler arquivos de entrada que contêm registros de comprimento fixo

Java MapReduce API – Conceitos Básicos

- OutputFormats
  - Determina como os dados são gravados nos arquivos
  - Exemplos de OutputFormats
    - FileOutputFormat: grava os dados de saída em um arquivo
    - DBOutputFormat: grava dados de saída em uma fonte de dados JDBC

Componentes de um programa MapReduce

- Um programa MapReduce típico contém os seguintes componentes:
  - Driver - código executado no cliente que configura e inicia a aplicação MapReduce
  - Mapper - classe Java que contém o método map ()
  - Redutor - classe Java que contém o método reduce ()

Driver

- Responsável por enviar a aplicação e sua configuração (**Job**) ao ResourceManager
- Os trabalhos podem ser enviados
  - De forma síncrona: aguarda a conclusão da aplicação antes de realizar outra ação
  - De forma assíncrona: não espera a conclusão da aplicação
- Pode configurar e enviar mais de uma aplicação
  - Por exemplo: fluxo de trabalho de aplicações MapReduce
- Parâmetros típicos
  - Caminho para dados de entrada
  - Caminho para dados de saída
  - Cluster HDFS / YARN a ser usado para a aplicação
- Implementado como uma classe Java contendo o ponto de entrada (método main ()) para o programa

Job Object

- Armazena a configuração do Job
  - Classes a serem usadas como Mapper e Reducer
  - Diretórios de entrada e saída
  - Nome do trabalho a ser exibido na UI do YARN Resource Manager

- Também pode ser usado para controlar o envio e a execução da aplicação e para consultar o seu estado

Mapper

- Classe Java contendo o método map ()
- Cada instância do Mapper itera por meio de seu InputSplit atribuído
  - Executa seu método map () para cada registro lido do InputSplit, usando um InputFormat definido e seu RecordReader associado
- O número de InputSplits (geralmente o número de blocos HDFS) nos dados de entrada determina o número de tarefas Map em um aplicativo MR
- Cuidados na implementação do método map
  - **Sem salvamento ou compartilhamento de estado** - os dados não devem ser compartilhados entre as tarefas Map
  - **Sem efeitos colaterais** - as tarefas Map podem ser executadas em qualquer sequência ou executadas mais de uma vez sem criar efeitos colaterais
  - **Nenhuma tentativa de se comunicar com outras tarefas Map** - as tarefas Map não podem se comunicar umas com as outras
  - **Cuidado ao executar operações de E/S externas (gravar em NFS ou acessar um serviço)** - pode ser percebido como um ataque DDoS ou tempestade de eventos

Reducer

- É executado sobre uma partição e respectivas chaves ordenadas
  - Seus valores associados são passados para o método reduce ()
- Os mesmos cuidados de implementação para mappers devem ser aplicadas ao implementar um reducer

Hello World no MapReduce → Word Count

- Conta a ocorrência de palavras em um arquivo texto
- Útil em problemas da vida real
  - Contar ocorrências de eventos específicos em arquivos de log
  - Funções de mineração de texto
  - **Nuvens de palavras**

- Entrada → arquivo texto
- Map → divide uma linha de texto em uma coleção de palavras (tokenização) e produz cada palavra como chave com um valor de 1
- Shuffle-and-Sort → agrupa os valores de cada palavra e classifica as chaves em ordem alfabética
- Reduce → soma os valores de cada palavra (chave)

WordCountDriver.java

```
public class WordCountDriver extends Configured implements Tool {
  public int run(String[] args) throws Exception {
   if (args.length != 2) {
    System.out.printf("Usage: %s [generic options] <inputdir> <outputdir>\n",
getClass().getSimpleName()); return -1; }
   Job job = Job.getInstance(getConf(), "Word Count");
   job.setJarByClass(WordCountDriver.class);
   FileInputFormat.setInputPaths(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));
   job.setMapperClass(WordCountMapper.class);
   job.setReducerClass(WordCountReducer.class);
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(IntWritable.class);
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   return job.waitForCompletion(true) ? 0 : 1;
 }
 public static void main(String[] args) throws Exception {
   int exitCode = ToolRunner.run(new Configuration(), new WordCountDriver(), args);
   System.exit(exitCode);
 }
}
```

WordCountMapper.java

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  private final static IntWritable one = new IntWritable(1);
  private Text wordObject = new Text();

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

   String line = value.toString();
   for (String word : line.split("\\W+")) {
    if (word.length() > 0) {
     wordObject.set(word);
     context.write(wordObject, one);
    }
   }
  }
}
```

WordCountReducer.java

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
  private IntWritable wordCountWritable = new IntWritable();
  @Override
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
              throws IOException, InterruptedException {
         int wordCount = 0;
         for (IntWritable value : values) {
             wordCount += value.get();
         }
         wordCountWritable.set(wordCount);
         context.write(key, wordCountWritable);
    }
}
```

Compilação, Empacotamento e Submissão

- Compilação
  - Todos os arquivos devem estar na mesma pasta

```
$ javac -classpath `hadoop classpath` *.java
```

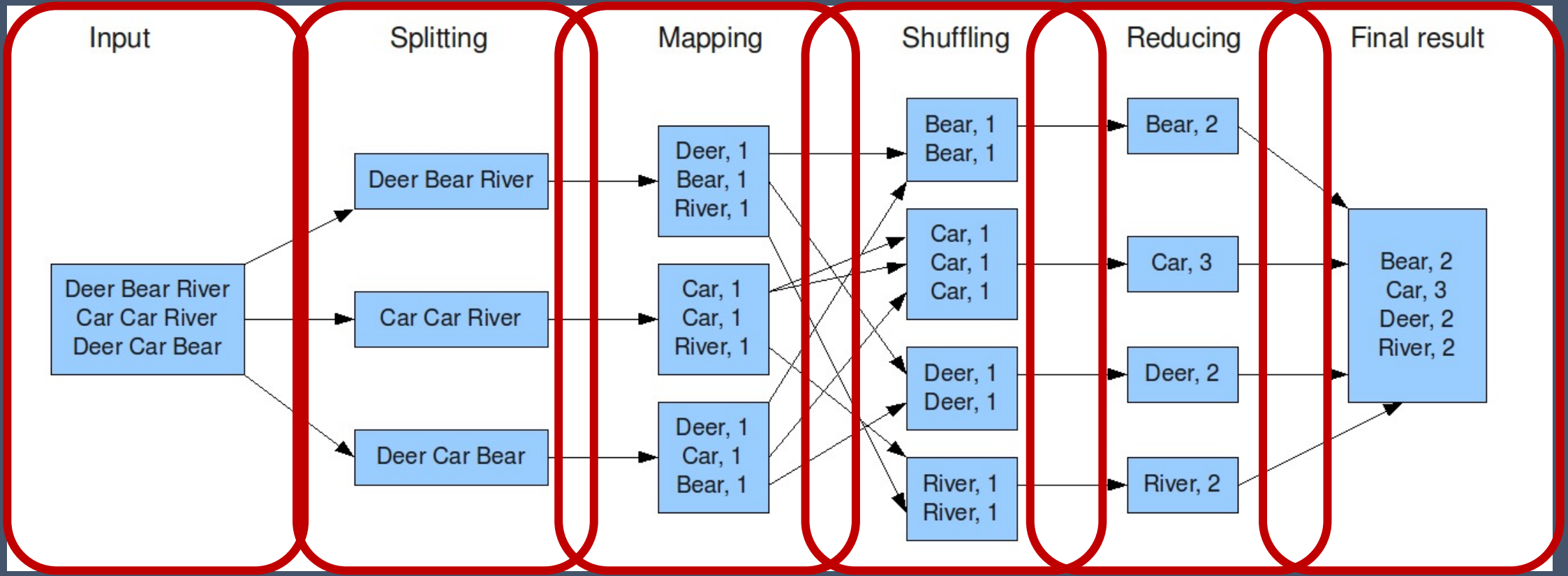
- Driver, Mapper e Reducer devem ser empacotados em um arquivo jar
  - Pode usar Maven ou javac / jar

```
$ jar cvf wc.jar *.class
```

- Submissão usando o comando hadoop jar

```
$ hadoop jar wc.jar WordCountDriver inputdir outputdir
```

Exemplo de execução – Wordcount



https://cs.calvin.edu/courses/cs/374/exercises/12/lab/

Conceitos avançados da API MapReduce

- Combiners
  - Diminui a quantidade de dados intermediários enviados entre mapeadores e redutores
  - Função combiner () é idêntica à função reduce ()
  - Os tipos da chave e valor de saída dp Map correspondem à entrada para o Combiner
  - Os tipos da chave e valor de saída do Combiner correspondem à entrada para o Reducer
  - A operação realizada deve ser comutativa e associativa
  - Declaração na classe Driver

```
job.setCombinerClass (WordCountReducer.class);
```

Conceitos avançados da API MapReduce

- Partitioners
  - Divide o “espaço de chaves” de saída para um programa MapReduce controlando os dados que cada Redutor obtém
  - Útil para distribuição de processos, balanceamento de carga ou separação da saída (ex. separando um arquivo para cada mês do ano)
  - HashPartitioner é usado por padrão
    - Usa uma função hash para separar o “espaço de chaves” em partes aproximadamente iguais entre os redutores
  - Declaração de um particionador personalizado na classe de driver

```
job.setPartitionerClass (MyCustomPartitioner.class);
```

  - Estende a classe base do Partitioner e tem um método getPartition ()

Conceitos avançados da API MapReduce

- Cache Distribuído
  - Usado para disseminar dados adicionais ou bibliotecas de classes em tempo de execução para mapeadores ou redutores em um ambiente de cluster Hadoop totalmente distribuído
  - DistributedCache envia os dados para todos os nós antes que qualquer tarefa seja executada
  - Os dados distribuídos estão disponíveis em formato somente leitura em qualquer nó que possui uma tarefa
  - Depois que a aplicação é encerrada, os arquivos são removidos automaticamente
  - Os itens podem ser adicionados usando a classe ToolRunner no Driver (argumentos -files, -archives, -libjars)
  - Exemplo: adicionar palavras irrelevantes na aplicação WordCount

```
$ hadoop jar wc.jar WordCountDriver -files stopwords.txt inputdir outputdir
```

  - Exemplo: acessando arquivos no DistributedCache

```
Arquivo f = novo arquivo ("stopwords.txt");
```

Hadoop Streaming

Hadoop Streaming

- Permite a implementação de funções Map e Reduce em linguagens diferentes de Java (por exemplo, Perl, Python, Ruby)
- Entrada e saída das fases Map e Reduce utilizam a entrada padrão (STDIN) e saída padrão (STDOUT)
- Algumas desvantagens
  - Apresenta sobrecarga adicional → o desempenho é pior do que a implementação em Java
  - Requer Java para implementar construções API adicionais (por exemplo, InputFormats, Writables, Partitioners)
  - Adequado apenas para dados representados como texto

Word Count em Python

```
#!/usr/bin/env python
# wordmapper.py
import sys
for line in sys.stdin:
  line = line.strip()
  words = line.split()
  for word in words:
   print '%s\t%s' % (word, 1)
```

```
#!/usr/bin/env python
# wordreducer.py
import sys

thisword = None
wordcount = 0
word = None
for line in sys.stdin:
  line = line.strip()
  word, count = line.split('\t', 1)
  count = int(count)
  if thisword == word:
    wordcount += count
  else:
    if thisword:
      print '%s\t%s' % (thisword, wordcount)
    wordcount = count
    thisword = word
if thisword == word:
  print '%s\t%s' % (thisword, wordcount)
```

Submetendo um Job Hadoop Streaming

```
$ hadoop jar \
  $HADOOP_HOME/share/hadoop/tools/lib/hadoop-
streaming-*.jar \
  -input inputdir \
  -output outputdir \
  -mapper wordmapper.py \
  -reducer wordreducer.py \
  -file wordmapper.py \
  -file wordreducer.py
```

mrjob

mrjob

- https://github.com/Yelp/mrjob
  - Última versão: 0.7.4 (setembro de 2020)

- Simplifica a escrita de código MR para Hadoop em Python
  - Com base no Hadoop Streaming
- Características
  - Suporta Python 2.7 / 3.4 +
  - Permite a escrita de tarefas MapReduce com várias etapas
  - Permite testar na máquina local
  - Permite executar em um cluster Hadoop ou cluster Hadoop na nuvem (Amazon EMR ou GCP DataProc)

Exemplo

mr_word_count.py

```
from mrjob.job import MRJob


class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

Execução (suporta diferentes modos)

```
# inline - single Python process
python mr_word_count.py -r inline input.txt

# local - simulates Hadoop (multiple processes)
python mr_word_count.py -r local input.txt

# hadoop - Hadoop cluster (default: stdout for output)
python mr_word_count.py -r hadoop hdfs:///input.txt
python mr_word_count.py -r hadoop hdfs:///input.txt \
    --output-dir hdfs:///output.txt

# emr - Amazon EMR
python mr_word_count.py -r emr s3://mydir/input.txt
```

Exemplo - WordCount

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")


class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)


if __name__ == '__main__':
    MRWordFreqCount.run()
```

Exemplo - multistep

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")


class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)
```

```
    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)


if __name__ == '__main__':
    MRMostUsedWord.run()
```

Ciências de Dados e Inteligência Artificial

Gerência de Infraestrutura para Big Data

MapReduce

Prof. Tiago Ferreto

tiago.ferreto@pucrs.br

PUCRS online UOL edtech_

Ciências de Dados e Inteligência Artificial

Gerência de Infraestrutura para Big Data

Pig

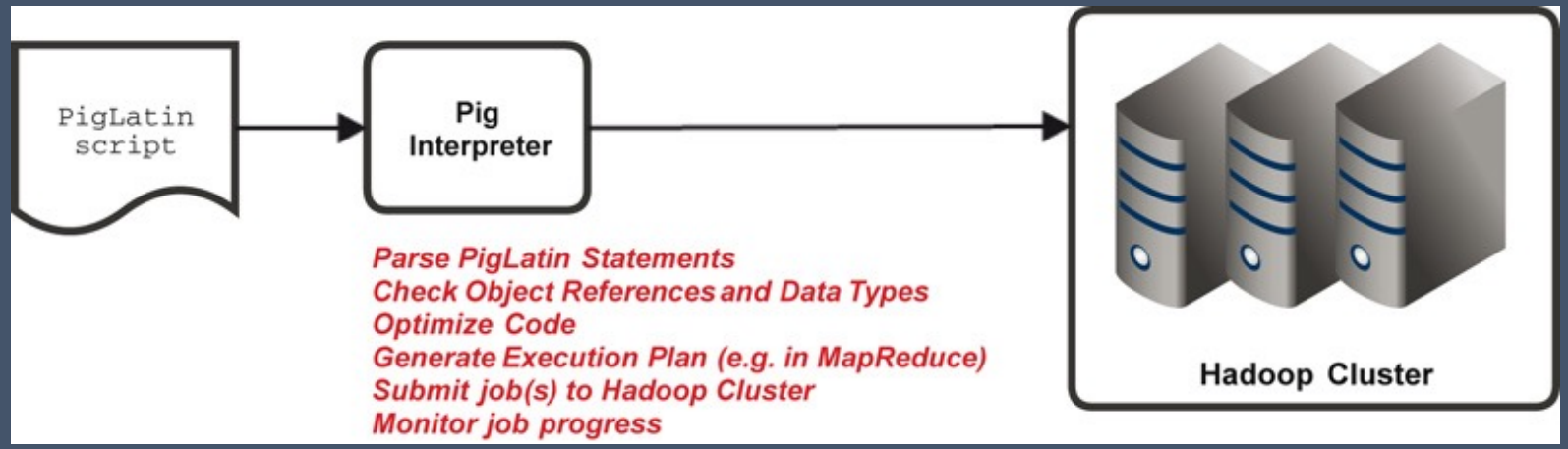
Prof. Tiago Ferreto

tiago.ferreto@pucrs.br

PUCRS online UOL edtech_

Pig - Introdução

- Projeto iniciado no **Yahoo!** em 2006 (tornou-se um projeto ASF em 2008)
- Objetivo: fornecer uma linguagem alternativa para programar MapReduce
- Abstrai o código MapReduce subjacente do desenvolvedor
  - Fornece uma **linguagem de fluxo de dados** (dataflow) interpretada , que é **convertida em uma série de** operações **Map/Reduce**
- **Pig Latin** → linguagem de script de fluxo de dados usada pelo Pig
- O intérprete (**Grunt**) recebe instruções PigLatin, transforma-as em tarefas MR, envia para o cluster, monitora seu progresso e retorna os resultados para o console ou salva-os em HDFS



Grunt – The Pig Shell

- Shell de programação **interativa**
- Usa **avaliação preguiçosa (*lazy evaluation*)**
  - As instruções são analisadas e interpretadas, mas a execução só começa quando a saída é solicitada
  - Permite planos de **otimização**
- Também pode ser executado em modo não-interativo ou em lote (batch)

Pig Latin

- Linguagem de fluxo de dados
  - Não é SQL ou linguagem orientada a objetos

- Um programa Pig Latin geralmente usa uma sequência de declarações com um padrão
  1. Carrega dados em um conjunto de dados nomeado
  2. Cria um novo conjunto de dados manipulando o conjunto de dados de entrada
  3. Repete a etapa 2 n vezes (conjuntos de dados intermediários não são necessariamente armazenados fisicamente)
  4. Envia o conjunto de dados final para o console ou diretório de saída

Pig – Estruturas de dados

- **Tuple**
  - Conjunto ordenado de campos
  - Cada campo pode ser outra estrutura de dados ou dados atômicos (átomo)
  - Exemplo: ('Jeff', 47)
- **Bag**
  - Coleção não ordenada de tuplas
  - Exemplo: {('Jeff', 47), ('Paul', 44)}
- **Map**
  - Conjunto de pares chave-valor
  - Exemplo: [nome # Jeff]

Identificadores de objeto

- Usado para identificar estruturas de dados (bags, maps, tuples e campos)
- Regras
  - Maiúsculas e Minúsculas
  - Não pode corresponder a uma palavra-chave do idioma Pig Latin
  - Deve começar com uma letra
  - Só pode incluir letras, números e sublinhados
- Exemplo: definir uma bag chamada 'alunos' com um esquema específico com identificadores para cada campo

```
students = LOAD 'student' AS (name: chararray, age: int,
gpa: float);
```

Pig – Tipos de dados simples (Atoms)

Type	Description	Example
int	32-bit signed integer	20
long	64-bit signed integer	20L
float	32-bit floating point	20.5F
double	64-bit floating point	20.5
chararray	UTF-8 string	'Jeff'
bytearray	uninterpreted byte array	*<BLOB>*
boolean	True or False	true
datetime	datetime	1970-01-01T00:00:00.000+00:00
biginteger	Java BigInteger	200000000000
bigdecimal	Java BigDecimal	33.4567833213234412334 42

Pig - Operadores relacionais e matemáticos

Operator	Notation
equal	==
not equal	!=
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=
pattern matching	matches
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%

Declarações, comentários e convenções no Pig

- As instruções são encerradas por ponto e vírgula (;)
  - Mas pode abranger várias linhas e incluir indentação
- As declarações começam atribuindo um conjunto de dados a uma bag
  - Carrega dados ou manipula uma bag previamente definida
  - Exceção: quando a saída é necessária (geralmente última linha - usando uma instrução DUMP ou STORE)
- Palavras-chave são capitalizadas por convenção (por exemplo, LOAD, STORE, FOREACH), mas não são case sensitive
  - No entanto, as funções integradas (por exemplo, COUNT, SUM) diferenciam maiúsculas de minúsculas
- Comentários

```
-- Inline comment
/*
Block comment
*/
```

Carregando dados no Pig

- Funções de carga → determinam como um esquema é extraído dos dados (semelhante a InputFormats)
  - A função de carga é definida através da cláusula USING
  - Se USING for usado, os **dados serão considerados dados de texto delimitados por tabulação**
- Funções comuns de carga do Pig
  - Outras funções de carga incluem: HBaseStorage, AvroStorage, AccumuloStorage, OrcStorage, etc

Function	Syntax	Description
PigStorage (default)	`PigStorage([delim])`	Loads and stores data as structured text files
TextLoader	`TextLoader()`	Loads unstructured data in UTF-8 format
JsonLoader	`JsonLoader([schema])`	Loads JSON data

Exemplo de instrução LOAD

- Instrução de carga para um conjunto de dados delimitado por vírgulas, terminado em nova linha

```
stations = LOAD 'stations' USING PigStorage(',') AS
            station_id,
            name,
            lat,
            long,
            dockcount,
            landmark,
            installation;
```

Pig Loading – schemas

```
--load data with no schema defined
stations = LOAD 'stations' USING PigStorage(',');
/* elements are accessed using their relative position, e.g. $0
all fields are typed as bytearray */

--load data with an untyped schema
stations = LOAD 'stations' USING PigStorage(',') AS
           (station_id,name,lat,long,dockcount,landmark,installation);
/* elements are accessed using their named identifier, e.g. name
all fields with an unassigned datatype are typed as bytearray */

--load data with a typed schema
stations = LOAD 'stations' USING PigStorage(',') AS
           (station_id:int, name:chararray, lat:float,
           long:float, dockcount:int, landmark:chararray,
           installation:chararray);
/* elements are accessed using their named identifier, e.g. name */
```

Instrução FILTER

- Filtra tuplas de uma bag
- Esquema não é alterado
  - As tuplas que correspondem aos critérios permanecem na bag, enquanto outras são descartadas
- Exemplo

```
sj_stations = FILTER stations BY landmark == 'San Jose';
```

Instrução FOREACH

- Equivalente a uma operação Map em um job MR
- Cada tupla em cada bag é iterada realizando uma operação solicitada
  - Exemplos de operações:
    - Remover um campo ou campos - projetar 3 campos de um bag com 4 campos
    - Adicionar um campo ou campos - adicione um campo computado com base nos dados da bag
    - Transformar um campo ou campos - converter uma string em minúsculas
    - Executar uma função de agregação - COUNT de um campo que é uma bag
- Exemplo

```
station_ids_names = FOREACH stations GENERATE station_id, name;
```

Instrução ORDER BY

- Ordena as tuplas por um determinado campo
- Exemplo

```
ordered = ORDER station_ids_names BY name;
```

Instrução DESCRIBE

- DESCRIBE - inspeciona o esquema de uma bag
  - Apresenta campos e tipos
  - Não invoca a execução das instruções anteriores
- Exemplo

```
DESCRIBE stations;
stations: {station_id: int,name: chararray,lat: float,long: float,
dockcount: int,landmark: chararray,installation: chararray}
```

Instrução ILLUSTRATE

- ILLUSTRATE – além de apresentar o esquema, também apresenta seus predecessores e dados de amostra (tuplas)
  - Requer execução - leva mais tempo para executar
  - Exemplo

```
ILLUSTRATE ordered;
-------------------------------------------------------------------------
| stations    | station_id:int   | name:chararray                    ...
-------------------------------------------------------------------------
|             | 23               | San Mateo County Center
|             | 70               | San Francisco Caltrain (Townsend at
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| station_ids_names   | station_id:int   | name:chararray              ...
-------------------------------------------------------------------------
|                     | 23               | San Mateo County Center
|                     | 70               | San Francisco Caltrain
-------------------------------------------------------------------------
-------------------------------------------------------------------------
| ordered    | station_id:int   | name:chararray                    ...
-------------------------------------------------------------------------
|            | 70               | San Francisco Caltrain (Townsend at
|            | 23               | San Mateo County Center
-------------------------------------------------------------------------
```

Funções integradas

- Normalmente operam em um campo e são usados com o operador FOREACH
- Exemplo

```
lcase_stations = FOREACH stations GENERATE station_id, LOWER(name),
                 lat, long, landmark;
```

- Funções comuns integradas

```
--Eval Functions
AVG, COUNT, MAX, MIN, SIZE, SUM, TOKENIZE

--Math Functions
ABS, CEIL, EXP, FLOOR, LOG, RANDOM, ROUND

--String Functions
STARTSWITH, ENDSWITH, LOWER, UPPER, LTRIM, RTRIM, TRIM, REGEX_EXTRACT

--Datetime Functions
CurrentTime, DaysBetween, GetDay, GetHour, GetMinute, ToDate
```

Instrução GROUP

- Permite agrupar registros por um campo específico
- Resulta uma estrutura com um registro por valor único no campo sendo usado para agrupar
- As tuplas na estrutura têm dois campos
  - Group: mesmo tipo de “campo de grupo”
  - Campo: nomeado com o nome do campo usado para agrupamento – contém um bag de tuplas desta relação que contém o elemento sendo agrupado
- Normalmente usado antes de executar uma função de agregação (COUNT, SUM, AVG, etc)
- Instrução GROUP ALL - agrupa todas as tuplas em uma estrutura
  - Pode ser contado usando COUNT ou COUNT_STAR (descarta valores NULL) ou somado usando SUM

```
--salespeople
--salespersonid, name, storeid
1,Henry,100
2,Karen,100
3,Paul,101
4,Jimmy,102
5,Janice,

--stores
--storeid, name
100,Hayward
101,Baumholder
102,Alexandria
103,Melbourne

--sales
--salespersonid, storeid, salesamt
1,100,38
1,100,84
2,100,26
2,100,75
3,101,55
3,101,46
4,102,12
4,102,67
```

```
grouped = GROUP sales BY salespersonid;
DESCRIBE grouped;
grouped: {group: int,
   sales: {(salespersonid: int,storeid: int,salesamt: int)}}
DUMP grouped;
(1,{(1,100,84),(1,100,38)})
(2,{(2,100,75),(2,100,26)})
(3,{(3,101,46),(3,101,55)})
(4,{(4,102,67),(4,102,12)})
salesbyid = FOREACH grouped GENERATE group AS salespersonid,
            SUM(sales.salesamt) AS total_sales;
DUMP salesbyid;
(1,122)
(2,101)
(3,101)
(4,79)
allsales = GROUP sales ALL;
DUMP allsales;
(all,{(4,102,67),(4,102,12),(3,101,46),(3,101,55),...})
--COUNT all tuples in the sales bag
sales_trans = FOREACH allsales GENERATE COUNT(sales);
DUMP sales_trans;
(8)
--SUM all salesamts in the sales bag
sales_total = FOREACH allsales GENERATE SUM(sales.salesamt);
DUMP sales_total;
(403)
```

<bag>.<field> notation = dereferencing operator

Instrução FOREACH aninhada

- Permite operar com o resultado de uma instrução GROUP
- O FOREACH aninhado repete cada item da bag aninhada
- Algumas restrições se aplicam
  - Somente operadores relacionais são permitidos na operação aninhada (LIMIT, FILTER, ORDER)
  - GENERATE deve ser a última linha no bloco de código aninhado

```
top_sales = FOREACH grouped {
            sorted = ORDER sales BY salesamt DESC;
            limited = LIMIT sorted 2;
            GENERATE group, limited.salesamt;};
DUMP top_sales;
(1,{(84),(38)})
(2,{(75),(26)})
(3,{(55),(46)})
(4,{(67),(12)})
```

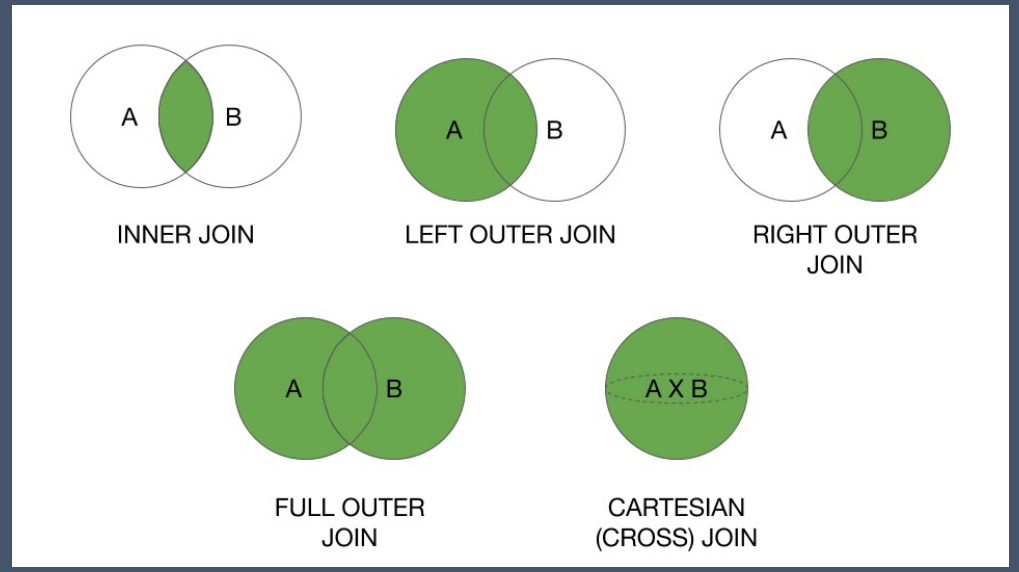
Instrução COGROUP

- Permite agrupar itens de várias bags
- O resultado contém uma bag para cada bag de entrada para a instrução COGROUP
- Exemplo

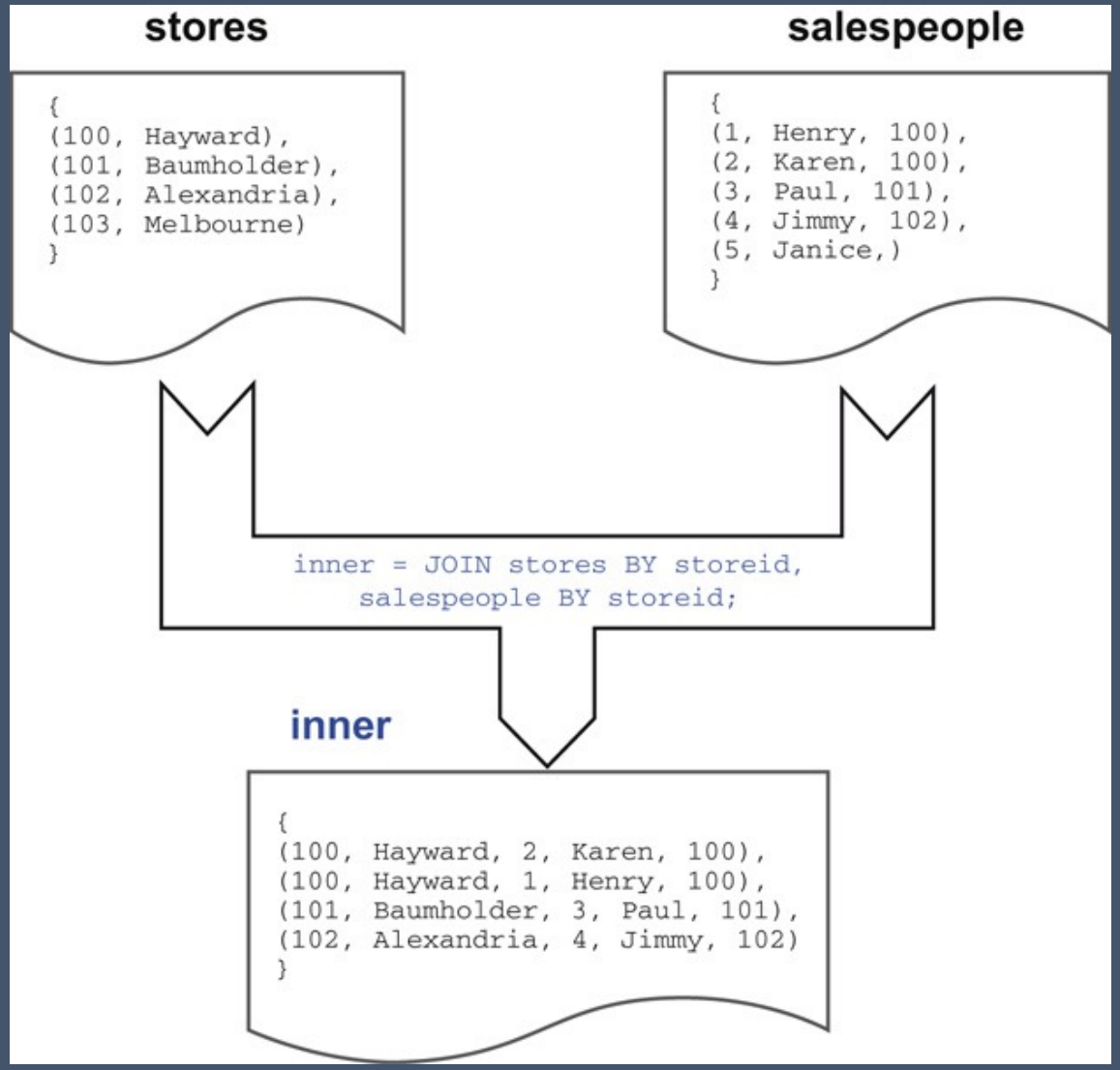
```
cogrouped = COGROUP stores BY storeid, salespeople BY storeid;
cogrouped: {group: int,stores: {(storeid: int,name: chararray)},
    salespeople: {(salespersonid: int,name: chararray,storeid: int)}}
DUMP cogrouped;
(100,{(100,Hayward)},{(2,Karen,100),(1,Henry,100)})
(101,{(101,Baumholder)},{(3,Paul,101)})
(102,{(102,Alexandria)},{(4,Jimmy,102)})
(103,{(103,Melbourne)},{})
(,{},{(5,Janice,)})
```

Instrução JOIN

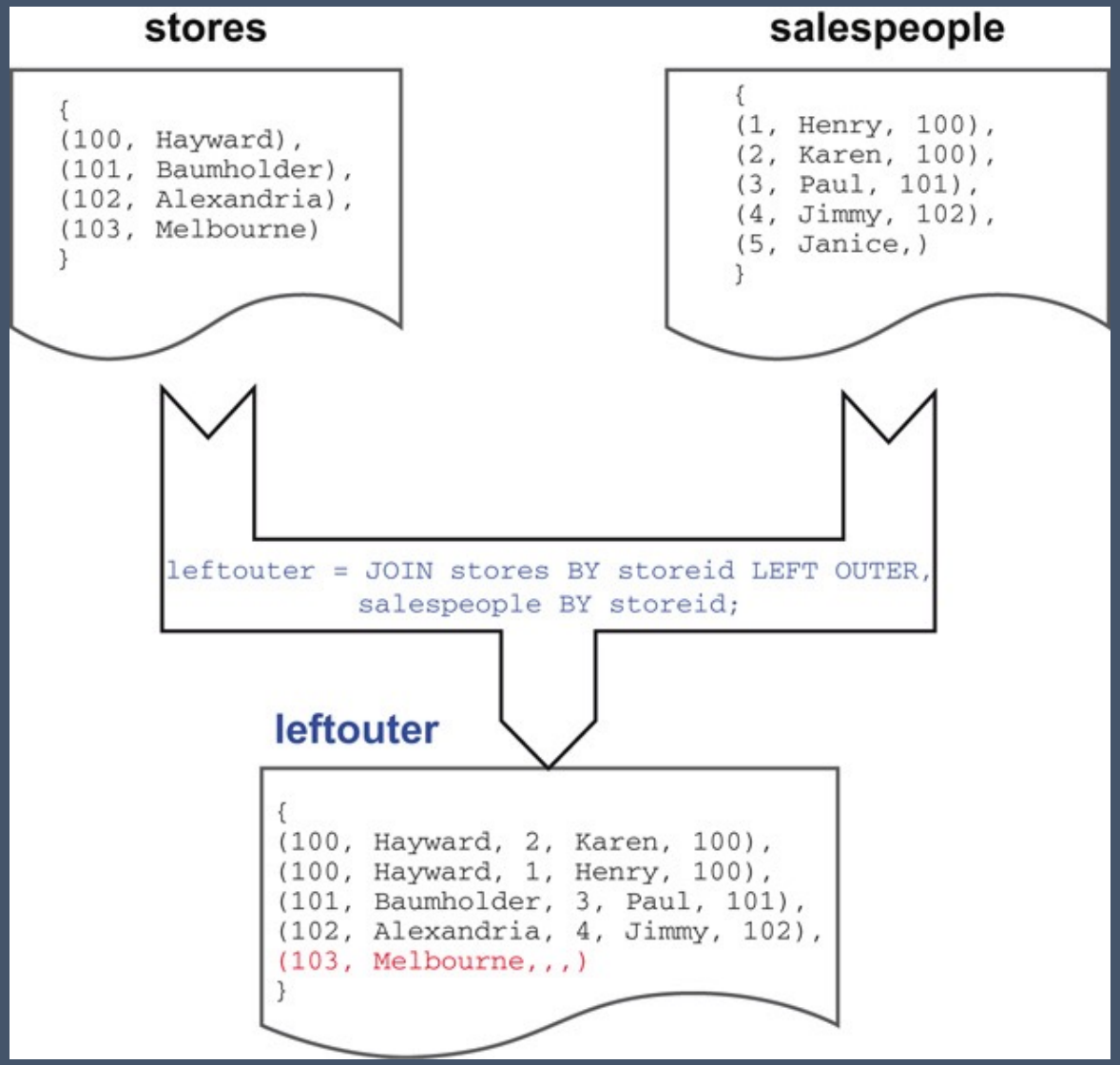
- Combina registros de duas bags com base em um campo comum (chave de junção)
- Tipos de junção
  - Inner join (ou simplesmente join) - retorna todos os registros de ambos os conjuntos de dados onde a chave está presente em ambos os conjuntos de dados
  - Left outer join - retorna todos os registros do primeiro conjunto de dados (esquerda) junto com os registros correspondentes apenas do conjunto de dados direito (direita)
  - Right outer join - retorna todos os registros do segundo conjunto de dados (direita) junto com os registros correspondentes apenas do primeiro conjunto de dados (esquerda)
  - Full outer join - retorna todos os registros de ambos os conjuntos de dados, haja uma correspondência de chave ou não



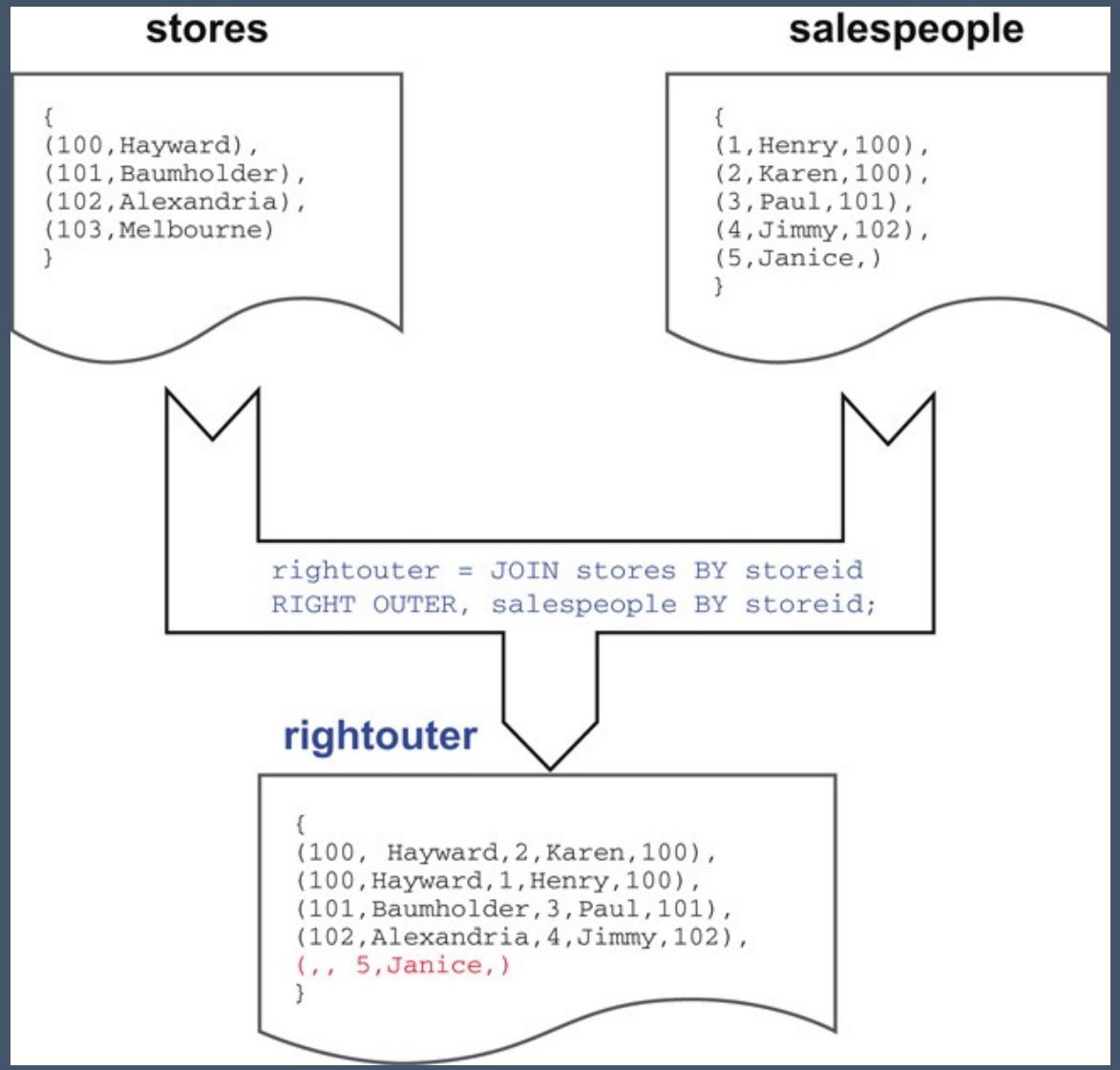
Inner Join no Pig



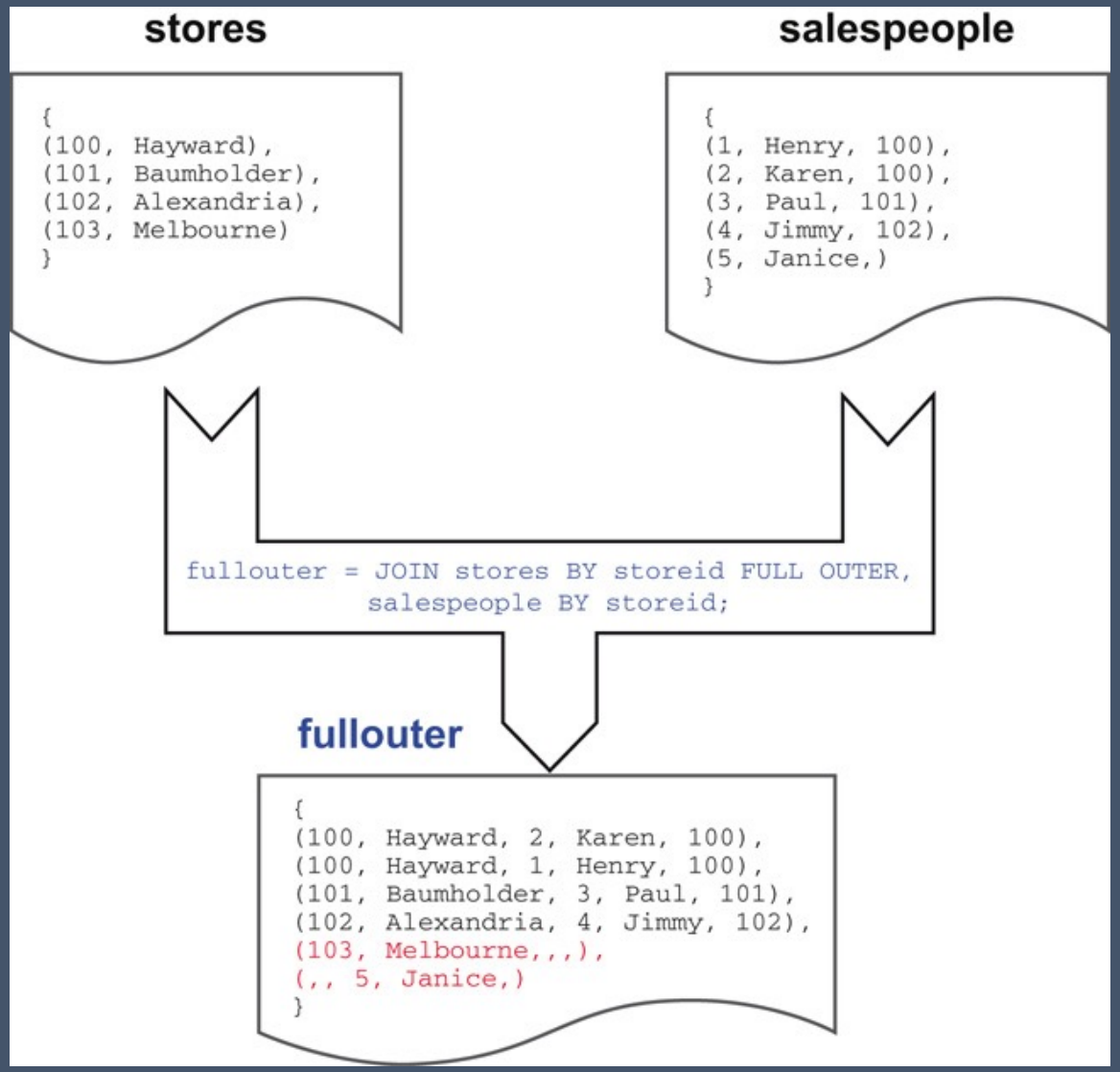
Left Outer Join no Pig



Right Outer Join no Pig



Full Outer Join no Pig



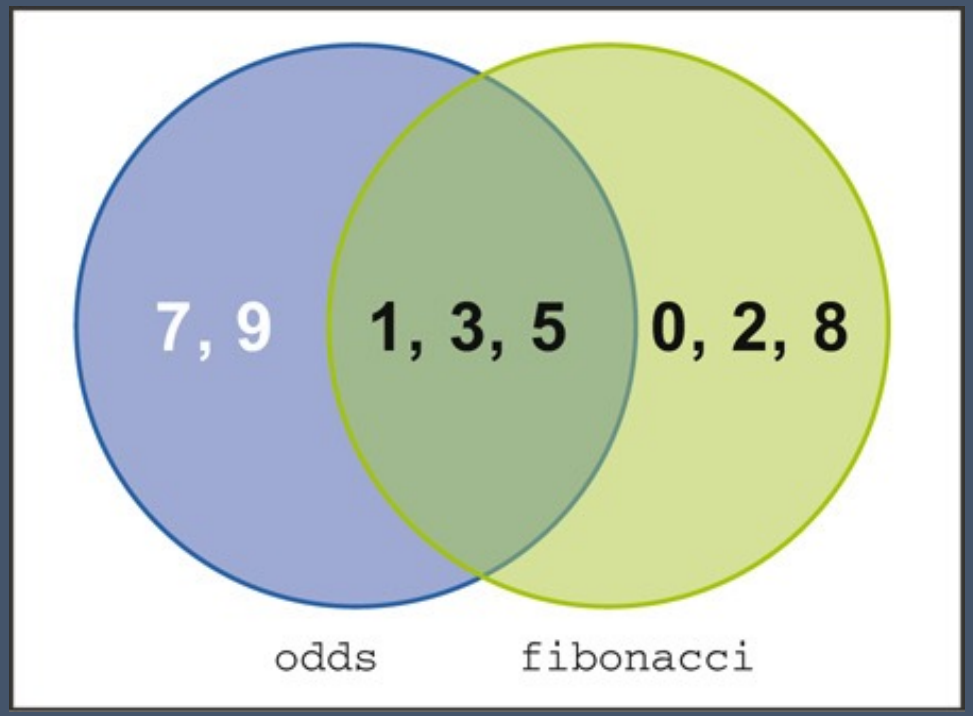
Recomendações

- 1º → Ao contrário de bancos de dados relacionais, não há otimizações para JOINs no Pig (sem índices ou estatísticas)
  - O desenvolvedor é responsável por otimizar manualmente a consulta
- Recomendação: “Juntar o grande pelo pequeno.” → referencie o maior dos dois bags de entrada primeiro seguido pelo menor

- 2º → Bags devolvidas pelas operações GROUP, COGROUP e JOIN irão conter campos duplicados
  - O campo no group, co-group ou join será duplicado na estrutura de dados resultante
- Recomendação: “Filtre antecipadamente, filtre com frequência”. → Seguir uma operação COGROUP com uma operação FOREACH para remover campos duplicados ajudará o otimizador Pig

Operações de conjunto

- O Pig suporta várias operações de conjunto comuns: UNION, DISTINCT, SUBTRACT, CROSS, SPLIT
- Exemplo com UNION




```
unioned = UNION odds, fibonacci;
DUMP unioned;
(0)
(1)
(2)
(3)
(5)
(8)
(1)
(3)
(5)
(7)
(9)
```

Funções definidas pelo usuário no Pig

- O Pig pode ser estendido por meio de UDFs (funções definidas pelo usuário)
- UDFs podem ser classificados em
  - Funções LOAD/STORE
    - O equivalente do Pig a InputFormats, OutputFormats e RecordReaders no MapReduce
    - Definir fontes e estruturas de dados de entrada e saída personalizadas
  - Funções de avaliação
    - Funções que podem ser usadas em expressões e instruções Pig para retornar qualquer tipo de dados Pig simples ou complexo
- UDFs podem ser escritas em várias linguagens: Java, Python, Jython, JavaScript, Groovy, Jruby
  - Plataformas JVM são recomendadas

PiggyBank

- Funções definidas pelo usuário contribuídas pela comunidade
  - https://cwiki.apache.org/confluence/display/PIG/PiggyBank

```
REGISTER 'lib/piggybank.jar';
DEFINE ISOToUnix
   org.apache.pig.piggybank.evaluation.datetime.convert.ISOToUnix();
...
toEpoch = FOREACH toISO GENERATE id,
           (long)ISOToUnix(ISOTime) as epoch:long;
...
```

Apache DataFu

- Biblioteca de UDFs Pig fornecidos pelo LinkedIn
- Inclui funções estatísticas e de processamento de conjuntos (não disponíveis nativamente ou no PiggyBank)
- http://datafu.apache.org/

```
REGISTER 'datafu-1.2.0.jar';
DEFINE Quantile datafu.pig.stats.StreamingQuantile('5');
quantiles = FOREACH resp_time_only {
    sorted = ORDER resp_time_bag BY resp_time;
    GENERATE url_date, Quantile(sorted.resp_time) as quantile_bins;
    }
...
```

Parametrizando Programas Pig

- Os parâmetros melhoram a flexibilidade do código
- Pig implementa parametrização por substituição de string
  - Os valores designados são substituídos por valores de parâmetros em tempo de execução

```
...
longwords = FILTER words BY SIZE(word) > $WORDLENGTH;
...
```

```
$ bin/pig -p WORDLENGTH=10 wordcount.pig
```

Ciências de Dados e Inteligência Artificial

Gerência de Infraestrutura para Big Data

Pig

Prof. Tiago Ferreto

tiago.ferreto@pucrs.br

PUCRS online UOL edtech_

PUCRS online UOL edtech_