MapReduce

Inteligência na Web e Big Data

Fabricio Olivetti de França e Thiago Ferreira Covões
folivetti@ufabc.edu.br, thiago.covoes@ufabc.edu.br

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Conceitos Básicos

Hadoop

Como vimos na aula inicial, muitas bases de dados de interesse prático necessitam de mais espaço do que os maiores HDs podem suportar.

Uma solução inicial é dividir os dados em vários HDs.

Hadoop

Exemplo: log de requisições http em um servidor de páginas de internet.

Dado um computador central que fará o gerenciamento das requisições e HDs externos conectados por rede.

Operacionalização: um HD é escolhido para armazenar o log atual. Ao atingir 98% da capacidade, outro HD é escolhido.

Hadoop

Quantos problemas vocês conseguem enumerar?

- Se um HD falha, perdemos aqueles dados para sempre.
- Se um HD falha, perdemos as amostras de todo um período, criando um viés estatístico.
- Se a comunicação com o HD atual falhar, o que fazer?
- Se o computador central falhar?

Hadoop Distributed File System

Criado para tratar essas questões de confiabilidade ao mesmo tempo que minimiza custos.

- Suporta arquivos muito grandes e gerencia milhares de nós ao mesmo tempo.
- Assume a possibilidade de lidar com hardware de baixo custo.
- Duplicação de arquivos para lidar com falhas.
- Detecção de falhas para previnir possibilidade de perdas.
- Computação otimizada: o processamento é feito onde os dados residem.
- Executa em máquinas e sistemas heterogêneos.

Arquitetura Geral

- Distribuído com um pouco de centralização.
- Nós principais: principais máquinas com poder alto de processamento e armazenamento.
- Nós principais gerenciam o envio e recebimento de tarefas de processamento para os outros nós (TaskTracker).
- Nós principais gerenciam os locais onde os dados devem residir, dados mais utilizados estão mais próximos (DataNode).
- Nós centrais mantém um mapa dos arquivos e diretórios do sistema distribuído (NameNode).

Arquitetura Geral

- Nós centrais envia as tarefas para os nós principais (JobTracker).
- Pensado em leituras frequentes de lotes de arquivos.
- Escrita de arquivo é custosa, geralmente *Write-Once, Read-Many*.
- Escrito em Java com suporte a Python.

NameNode

- Armazena os metadados típicos de um sistema de arquivo.
- Apenas um servidor armazenando o NameNode, ele deve ser o mais importante, estável e seguro.
- Cuida da criação de réplicas de blocos sempre que ocorre falha em um DataNode.

DataNode

- Armazena os dados de arquivos.
- Suporta qualquer sistema de arquivo (FAT, NTFS, ext, etc.).
- Notifica o NameNode sobre os blocks que ele possui (ao substituir um NameNode ele requisita tal informação).
- Arquivos são armazenados em blocos de 64MB por padrão.
- Envia um relatório periódico ao NameNode.
- Envio de dados inteligente, tem preferência pelo envio aos DataNodes mais próximos.

Estratégias de Armazenamento

- O NameNode réplica cada bloco de arquivo 2 vezes em um *rack* local e uma vez em outro *rack*.
- Réplicas adicionais podem ser distribuídos aleatoriamente para outros nós.
- Ao requisitar um certo bloco, esse é recuperado do nó mais próximo ao cliente.
- Em caso de falha (falta de relatório periódico), o NameNode escolhe outros DataNodes para replicar.
- Otimiza o balanceamento do armazenamento e comunicação de rede.

Estratégias de Corretude

- Usa CRC32 para validar os dados.
- Calcula checksum para cada 512 bytes de dados, DataNode armazena o crc.
- Cliente recebe os dados e seus respectivos checksums.
- Em caso de falha de verificação, cliente reporta e recebe de outra réplica.

Comandos básicos

- hadoop dfs -mkdir /diretorio
- hadoop dfs -cat /diretorio/arquivo.txt
- hadoop dfs -rm /diretorio/arquivo.txt

Conta com interface Web:
`http://host:port/dfshealth.jsp`

MapReduce

Modelo MapReduce

Modelo de programação distribuída.

Dean, Jeffrey, and Sanjay Ghemawat. ”MapReduce: simplified data processing on large clusters.”Communications of the ACM 51.1 (2008): 107-113.

Ideias: Unix pipeline e básico de Programação Funcional:

Modelo MapReduce

```
cat input | grep | sort | uniq -c | cat > output
input > map > shuffle & sort > reduce > output
```

Trabalha com *stream* de dados.

MapReduce

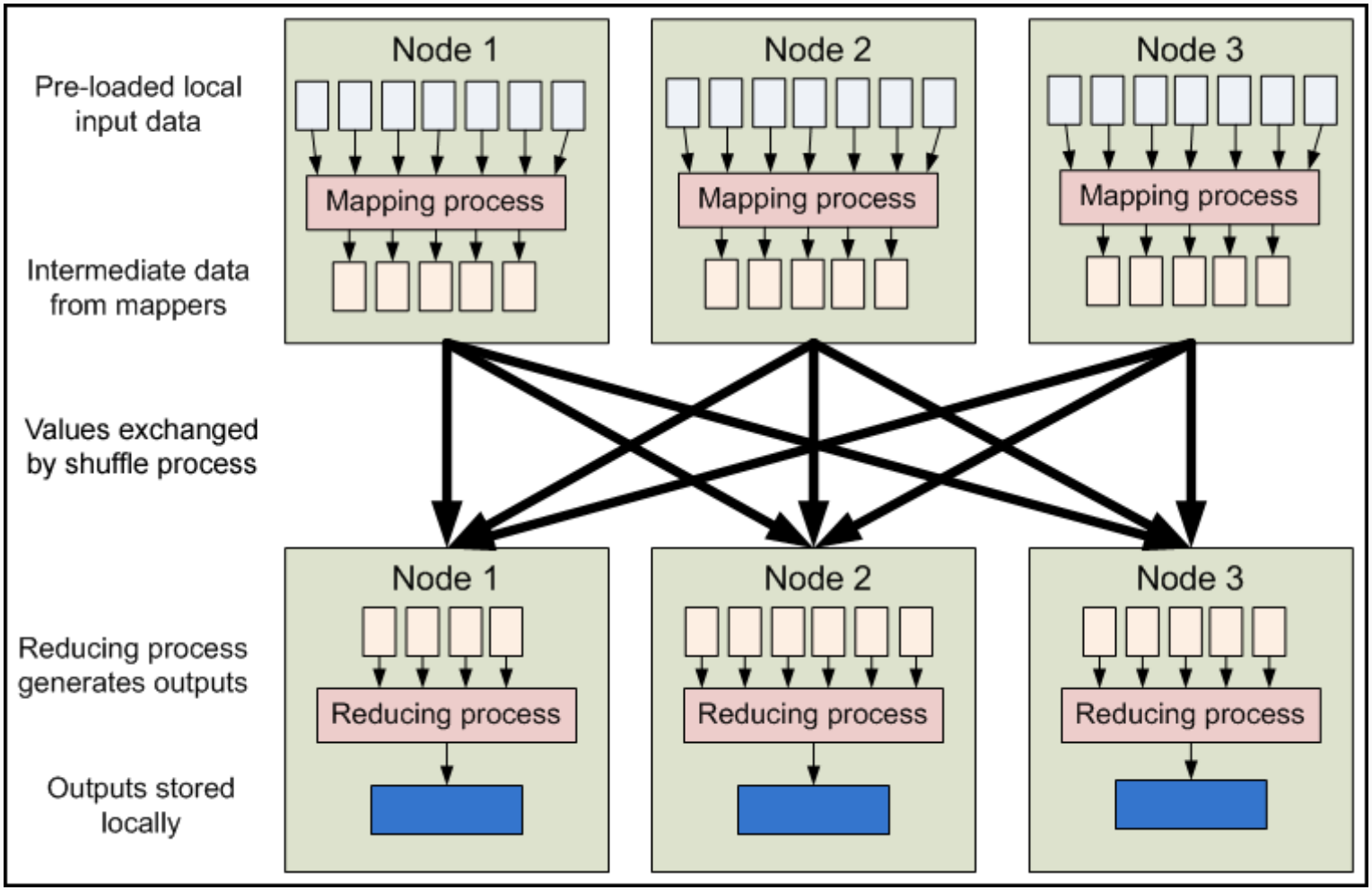

Figura 1: Fonte: UT Dallas

Modelo MapReduce

- Balanceamento de carga otimizado.
- Recuperação rápida de falhas.
- Possibilidade de reprocessar em caso de falhas.
- Processamento duplicado em caso de lentidão.
- Otimizações locais.

Modelo MapReduce

Modelo extremamente simples composto de duas funções:

- `Mapper`, equivalente ao nosso `map`
- `Reducer`, equivalente ao nosso `fold`

Modelo MapReduce

As assinaturas das funções passa a ser:

```
1 mapper(key, value) -> (key, value)
2 reducer(key, Iter[value]) -> (key, value)
```

Modelo MapReduce

O usuário apenas define as funções que devem ser passadas ao `Mapper` e ao `Reducer` e o sistema cuida de todo o resto.

Modelo MapReduce

Por outro lado, a simplicidade leva ao problema de reescrever diversos algoritmos utilizando apenas esses dois componentes.

Não permite muita flexibilidade…

Modelo MapReduce

Dentre os desafios, não temos informação de:

- Em quais nós os processos estão sendo executados
- Quando cada processo inicia e termina
- Quais pares de chave-valor estão sendo processados por um certo `mapper`
- Quais pares chave-valor intermediários estão sendo processados por um certo `reducer`

Modelo MapReduce

Por outro lado:

- Não existe limitação quanto a estrutura usada como chave e valor
- Hadoop permite execução de código de inicialização e término para as tarefas de `mapper` e `reducer`
- Hadoop permite especificar a ordenação e particionamento das chaves, para garantir um certo grupo esteja no mesmo nó

Word Count

- Mapper
  - Entrada: linhas de texto
  - Saída: chave = palavra, valor = 1
- Reducer
  - Entrada: chave = palavra, valor = lista de contagens
  - Saída: chave = palavra, valor = soma

MapReduce

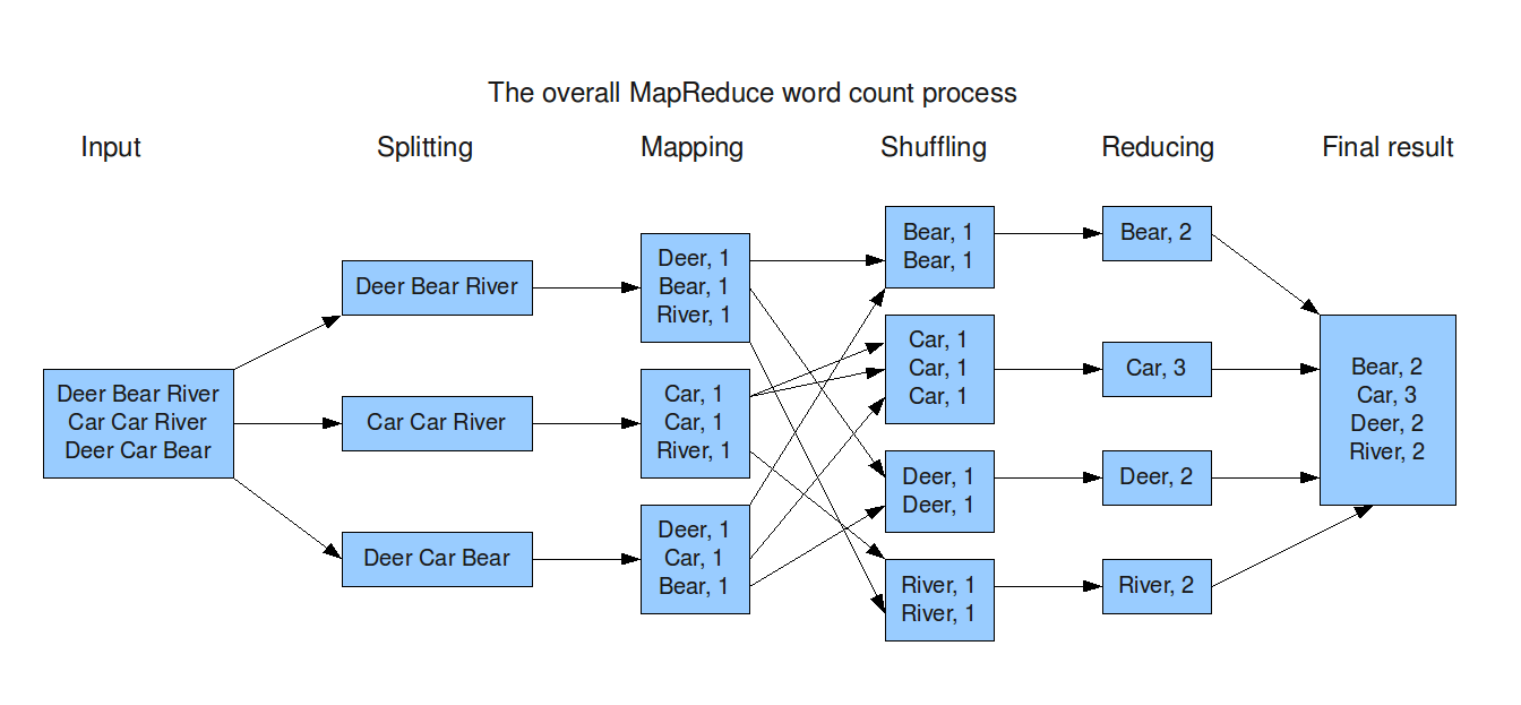

Figura 2: Fonte: UT Dallas

MapReduce

```
1  def mapper(key : int, words : str) -> (str, int):
2    for word in words:
3      yield (word, 1)
4
5  def reducer(key : str, values : [int]) -> (str, int):
6    yield (key, sum(values))
```

Combiners

Uma forma de minimizar a carga dos Reducers, gerando uma avaliação parcial em cada nó.

É executado localmente, geralmente é igual ao próprio reducer.

Combiners

```
def mapper(key : int, token : str) -> (str, int):
  counts = defaultdict(int)
  for word in token.split():
    counts[word] += 1
  for k, v in counts.iter():
    yield (k, v)
```

Calculando a média

Tarefa: calcular a média dos valores associados a cada chave.

Calculando a média

```
1  def mapper(key : str, value : int) -> (str, int):
2    yield key, value
3
4  def reducer(key : str, values = [int]) -> (str, int):
5    yield key, sum(values)/len(values)
```

Calculando a média

Se tentarmos fazer um *combiner* para esse algoritmo cometeríamos um grande erro:

> A média das médias de agrupamentos de um conjunto de números é diferente da média desse conjunto

Calculando a média

Podemos utilizar a função `combiner` que é executada localmente em cada nó após o `mapper`, servindo como um pré-processamento do `reducer`.

Calculando a média

```
1  def mapper(key : str, value : int) -> (str, int):
2    yield key, value
3
4  def combiner(key : str, values : [int]) -> (str,
   ↪  (int,int)):
5    yield key, (sum(values), len(values))
6
7  def reducer(key : str, values = [(int, int)]) -> (str,
   ↪  int):
8    sums = sum(map(fst, values))
9    lens = sum(map(snd, values))
10   yield key, sums/lens
```

Calculando a média

O código anterior apresenta um problema: não temos garantia que o `combiner` será realmente executado. No caso em que ele não é, o `reducer` irá receber um tipo de entrada diferente do esperado!

Calculando a média

Para resolver esse problema podemos fazer:

```
1  def mapper(key : str, value : int) -> (str, int):
2    yield key, (value, 1)
3
4  def combiner(key : str, values : [(int, int)]) -> (str,
   ↪  (int,int)):
5    sums = sum(map(fst, values))
6    lens = sum(map(snd, values))
7    yield key, sums/lens
8
9  def reducer(key : str, values : [(int, int)]) -> (str,
   ↪  int):
10   sums = sum(map(fst, values))
11   lens = sum(map(snd, values))
12   yield key, sums/lens
```

Trabalhando com tipos de dados

No exercício anterior utilizamos a estrutura de *tuplas* para permitir o uso de `combiner` para o nosso problema.

Existem dois padrões de programação bastante comuns quando otimizando algoritmos para o conceito de *MapReduce*.

Co-ocorrência de palavras

Vamos considerar o problema de contar o número de co-ocorrências de cada par de palavras em um corpus.

Em um corpus com n palavras, esse algoritmo gera uma matriz $n \times n$ em que o elemento c_{ij} representa a frequência de co-ocorrência da palavra i com a palavra j.

Co-ocorrência de palavras

Uma primeira alternativa para o algoritmo é usando a nossa estratégia de *tuplas*:

```
1  def mapper(key : int, value : str) -> ((str, str), int):
2    words = tokenize(value)
3    for i in range(len(words)):
4      for v in neighbors(words[i]):
5        yield ((words[i], v), 1)
6
7  def reducer(key : (str, str), values : [int]) ->
   ↪  ((str,str), int):
8    yield (key, sum(values))
```

Co-ocorrência de palavras

Uma outra estratégia, chamada de *stripes* utiliza uma array associativa para reduzir ainda mais o trabalho do `reducer`:

```
1  def mapper(key : int, value : str) -> (str, dict):
2    words = tokenize(value)
3    for i in range(len(words)):
4      counter = defaultdict(int)
5      for v in neighbors(word[i]):
6        counter[v] += 1
7      yield (words[i],counter)
8
9  def reducer(key : str, values : [dict]) -> (str, dict):
10   counter = defaultdict(int)
11   for v in values:
12     counter = union(counter, v)
13   yield key, counter
```

Co-ocorrência de palavras

Ambas soluções podem se beneficiar de um `combiner`, pois as operações efetuadas são associativas e comutativas.

Co-ocorrência de palavras

Porém, a versão *stripe* pode se beneficiar mais uma vez que a chave é apenas um termo, ou seja, ela sempre terá oportunidade de agregar informação quando um mesmo termo aparecer múltiplas vezes no documento.

Co-ocorrência de palavras

Já na versão com *tuplas*, a chave é uma combinação de duas palavras, e a probabilidade de um par de palavras co-ocorrer em um documento é menor do que a probabilidade de uma delas ocorrer.

Co-ocorrência de palavras

Por outro lado é preciso tomar cuidado com o crescimento do uso de memória da versão *stripe*, para palavras muito comuns o tamanho da array associativa pode se tornar proibitivo.

Co-ocorrência de palavras

Uma solução intermediária é criar chaves com a estrutura `((str, char), dict)` de tal forma que a chave passa a ser a palavra que contaremos as co-ocorrências, um caractere sinalizando um *bucket* e o valor será a array associativa contendo todas as palavras começando por esse caractere.

Co-ocorrência de palavras

Com isso reduzimos nossas arrays intermediárias em pedaços menores, tornando possível o restante das otimizações induzida por ela.

Comentários Finais

Comentários Finais

Vimos na aula de hoje como funciona um dos sistemas de arquivos distribuídos mais conhecidos.

Além disso aprendemos como implementar alguns algoritmos simples no contexto de *MapReduce*.

Além de algoritmos básicos, aprendemos também sobre o uso das técnicas de *tuplas* e *stripes*.