

NoSQL MongoDB MapReduce Cassandra  
Python Processamento  
BerkeleyDB Paralelismo DFS  
chave-Valor Documentos Colunar

Paradigmas de computação  
distribuída e bancos de  
dados

---

# Paradigmas de computação e bancos de dados

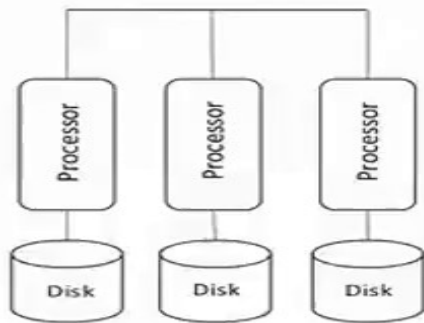
## Agenda

- Sistemas de arquivos distribuídos
- Algoritmo *Map Reduce*
- Consistência em bancos de dados não relacionais

# Paradigmas de computação e bancos de dados

Muitas soluções NoSQL buscam:

- Resolver problemas *Big Data*;
- Demandar estrutura de computação paralela;
- Esquemas de particionamentos (*shared nothing*);



- Tolerância a falhas;
- Escalabilidade

# Paradigmas de computação e bancos de dados

Alguns paradigmas de computação são fundamentais:

- Várias soluções vão utilizar sistemas de arquivos distribuídos;
- Esquemas de controle de réplicas e consistência dos dados;
- Algoritmos para execução paralela em conjunto com estruturas de arquivos;

NoSQL MongoDB MapReduce Cassandra  
Python Processamento  
BerkeleyDB Paralelismo DFS  
chave-Valor Documentos Colunar

Paradigmas de computação  
distribuída e bancos de  
dados

---

Sistemas de arquivos  
distribuídos

# Sistemas de arquivos distribuídos

## Histórico:

- Processamento intensivo era feito em hardware especializado (processadores, cache, discos e memória);
- A Web e o *Big Data* exigem processamento intensivo, mas em outra estrutura de hardware:
  - Centenas ou milhares de computadores em rede (nós);
  - Operação destes computadores de forma mais ou menos independente;
  - Cada um dos nós é um '*commodity hardware*' – custo reduzido;
  - A estrutura em geral é tolerante a falhas;
  - Utilizam sistemas de arquivos especializados;

# Sistemas de arquivos distribuídos

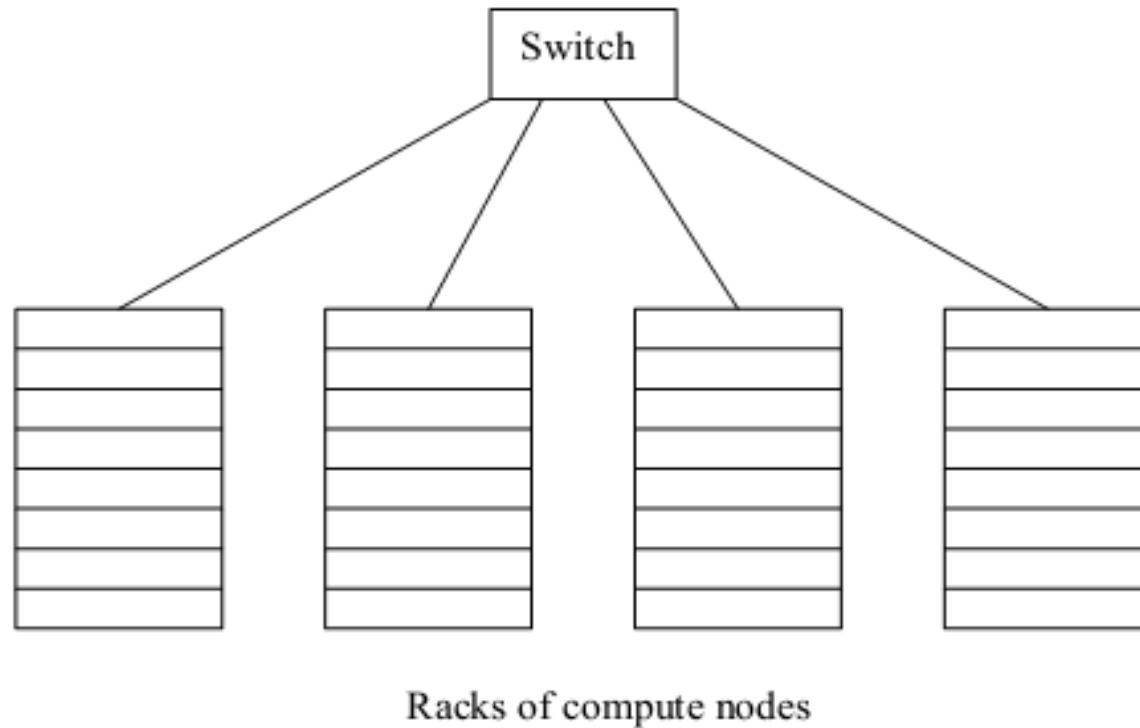
## Organização física da estrutura:

- A organização física destas máquinas pode seguir este exemplo:
  - Nós são armazenados em *racks* (8-64 em um *rack*);
  - Os nós em um *rack* são conectados via rede (*gigabit Ethernet*);
  - Conjuntos de *racks* são disponíveis na estrutura formando uma espécie de *cluster*;
  - A conexão entre os *racks* também pode ser otimizada;
  - Quanto maior o número de *racks* ou nós, maior a probabilidade falha (de um dos nós);

# Sistemas de arquivos distribuídos

## Organização física da estrutura:

- A organização física destas máquinas pode seguir este exemplo:





# Sistemas de arquivos distribuídos

## Computação nesta estrutura:

- Cálculos computacionais nesta estrutura podem levar minutos ou mesmo horas;
- Os cálculos não podem ser reiniciados toda vez que um componente (*rack* ou nó de execução) falha;
- Proposta de solução:
  - Arquivos armazenados de forma redundante (*Distributed File System* - DFS);
  - Cálculos devem ser divididos entre os nós, de forma que se algum nó falhar, somente o trabalho atribuído ao nó deve ser reexecutado;

# Sistemas de arquivos distribuídos

## Características DFS:

- Arquivos devem ser 'grandes', gigabytes, pelo menos; Arquivos menores não fazem sentido no *DFS*;
- Arquivos no *DFS* são raramente atualizados (*write-once-read-many*). Adicionalmente dados são adicionados para os arquivos (periodicidade, processamento batch);
- Arquivos são divididos em partes ('*chunks*' ou blocos), normalmente 64 megabytes e replicados para, pelo menos, 3 nós (em *racks* diferentes);
- Os Itens acima são customizáveis;

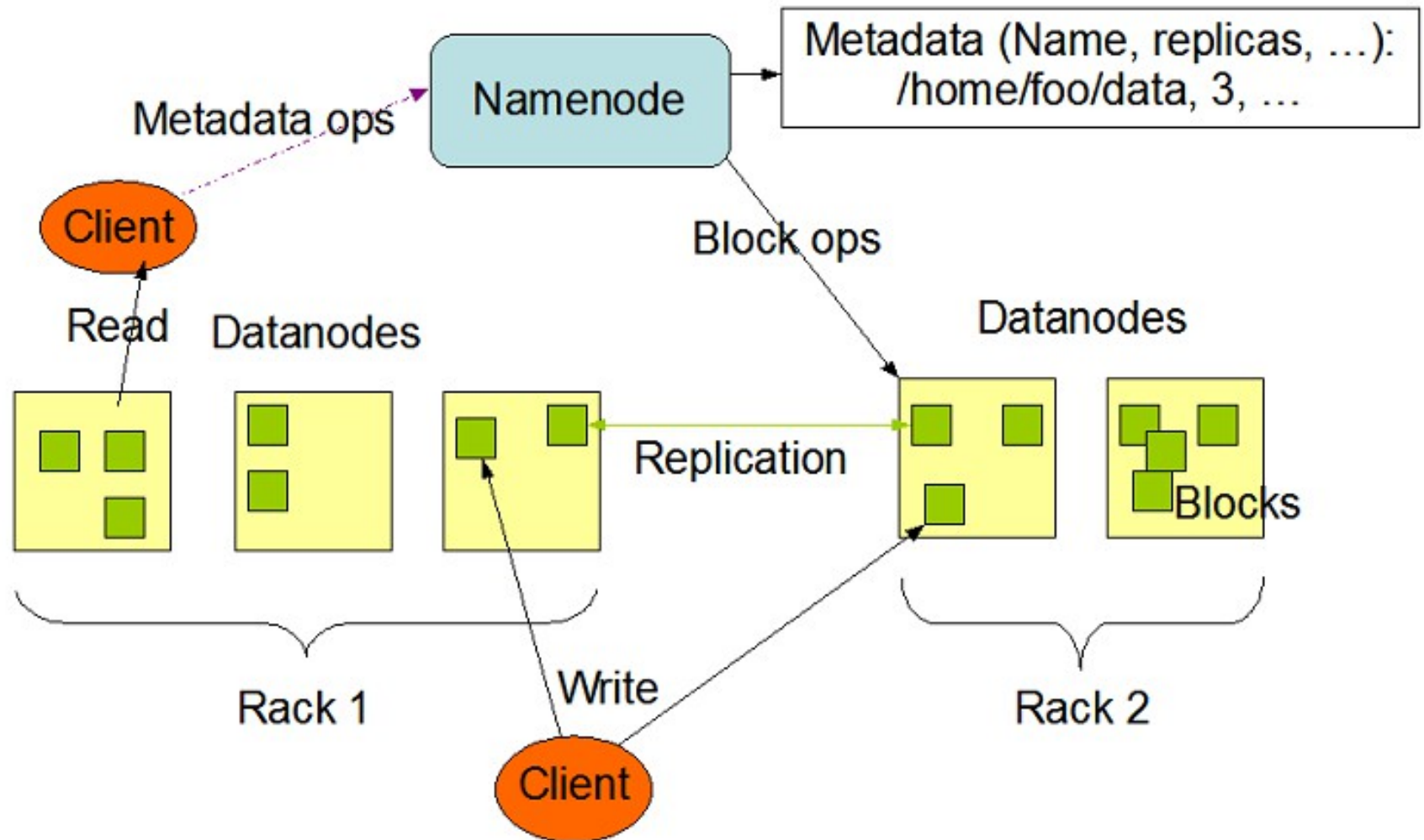
# Sistemas de arquivos distribuídos

## Características *DFS*:

- As informações dos blocos e replicas é controlado utilizando metadados e com um figura central no cluster: '*name node*' ou '*master node*';
- *Name node*:
  - Gerencia o sistema de arquivos(réplicas, blocos, nós e *racks*): abrir, fechar, renomear arquivos;
  - Gerencia o acesso dos clientes ao arquivos;
- Os outros nós do *cluster* são chamados de '*data node*' ou '*slave node*';
  - Executam as operações enviados pelo '*Name node*': criação, exclusão e replicação de blocos;

# Sistemas de arquivos distribuídos

## Características *DFS*:



# Sistemas de arquivos distribuídos

## Características DFS:

- Possuem regras de sistemas de arquivos: *rack*, '*data node*', *namespaces*, diretórios e arquivos;
- Além disto o DFS gerencia os blocos e sua distribuição/replicação nos '*data nodes*';
- Padrão de réplicas 1/3(fora do rack) e 2/3(no rack);
- O '*name node*' periodicamente recebe um relatório de blocos do '*data node*';

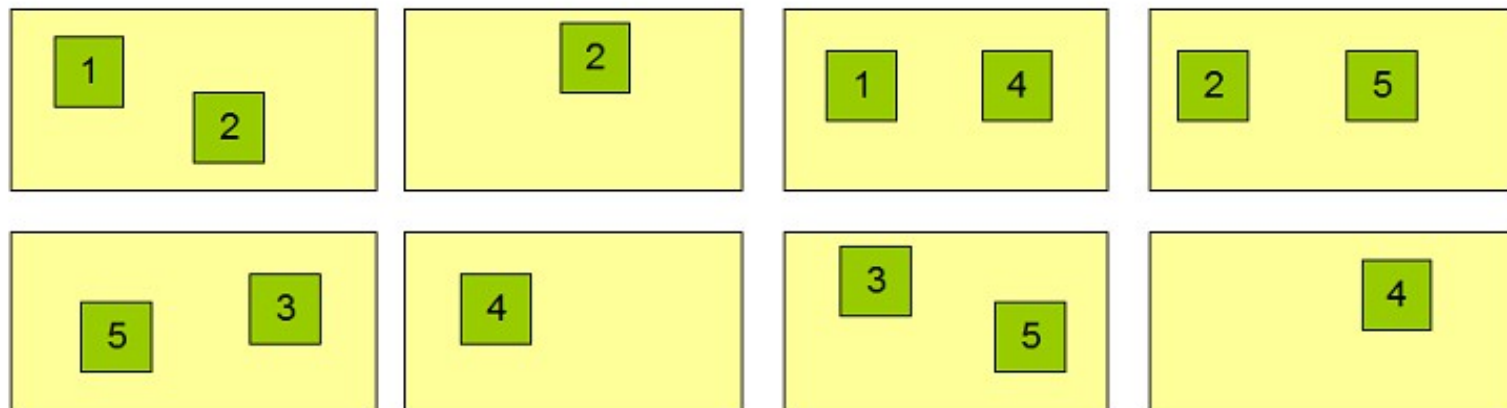
# Sistemas de arquivos distribuídos

## Características DFS:

### Block Replication

Namenode (Filename, numReplicas, block-ids, ...)  
/users/sameerp/data/part-0, r:2, {1,3}, ...  
/users/sameerp/data/part-1, r:3, {2,4,5}, ...

### Datanodes



# Sistemas de arquivos distribuídos

## Características DFS:

- O '*name node*' é replicado para outra estrutura no *DFS* e também possui *log* de alterações;
- A estrutura de arquivos é, tipicamente, mantida em memória;
- '*Name node*' também faz rebalanceamento de carga: elevada demanda para um bloco, por exemplo;

# Sistemas de arquivos distribuídos

## Implementações DFS:

- '*Google File System*' (*GFS*), implementação original e primeira a ficar 'famosa';
- *Hadoop Distributed File System* (*HDFS*), *open-source*. É disponibilizada pela fundação apache e implementada em java;
- CloudStore, é outra implementação *DFS open-source*, originalmente desenvolvida pela Kosmix.



NoSQL MongoDB MapReduce Cassandra  
Python Processamento  
BerkeleyDB Paralelismo DFS  
chave-Valor Documentos Colunar

Paradigmas de computação  
distribuída e bancos de  
dados

---

Algoritmo Map Reduce

# Algoritmo Map Reduce

## Origens:

- Patente original é do Google, mas é utilizado em várias outros sistemas de computação paralela;
- A ideia é derivada da programação funcional:
  - *Map e reduce* são dois tipos de funções comuns;
  - Map:
    - Aplica um função ou operação para cada elemento em uma lista;  
Ex.: multiplicação por 2;  
[1,2,3,4] Map function → [2,4,6,8,]
    - Não altera o dado original. Evita o principio '*Shared Data*';
    - Pode ser executado de forma paralela;

# Algoritmo Map Reduce

## Origens:

- A ideia é derivada da programação funcional:
  - *Reduce*:
    - É uma função de agrupamento ou compressão;
    - Aplica uma função em conjunto de dados reduzindo para um simples valor;
    - Pode ser executado de forma paralela;
    - Ex.: [2,4,6,8,] → Reduce function → [20]
- De forma geral:
  - O algoritmo pode ser usado sempre que houver uma lista;
  - Para cada elemento da lista uma função que a transforme;
  - Outra função que possa ser aplicada ao conjunto de dados transformados de forma a agregá-los;

# Algoritmo Map Reduce

## Detalhes de funcionamento:

- A implementação do algoritmo é utilizada para realizar computação no *DFS* para arquivos 'grandes' e com execução tolerante a falha;
- É necessário escrever as duas funções: *Map* e *reduce*;
- O sistema lida com os demais detalhes:
  - Execução paralela;
  - Coordenação de tarefas (*Map* e *reduce*);
  - Lidar com a tolerância a falhas;

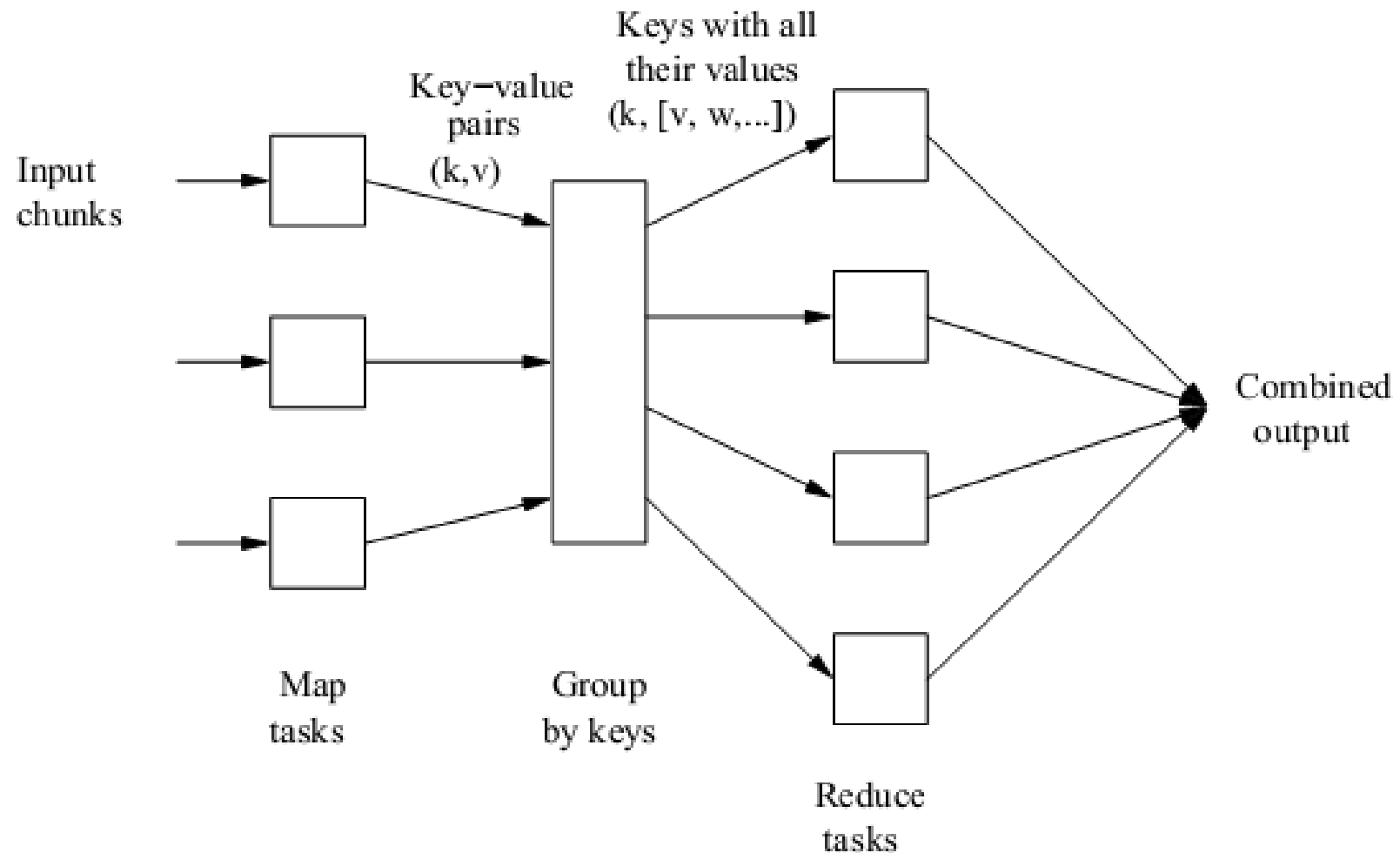
# Algoritmo Map Reduce

## Detalhes de funcionamento:

- Seguintes passos de execução:
  - Um arquivo é armazenado no *DFS* com vários blocos em vários nós e *racks*;
  - Um conjunto de tarefas do tipo *Map* é criado, para cada *Map* existe um ou mais blocos que serão processados; As tarefas *Map* vão transformar o dado em um estrutura chave valor ou tuplas;
  - As estruturas chave valor são coletadas pelo controlador *master* e ordenadas pelas suas chaves;
  - As chaves serão agrupadas e divididas para as tarefas do tipo *Reduce* (uma chave, com vários valores será processado por uma e só uma tarefa *Reduce*);
  - As tarefas do tipo *Reduce* vão então agrupar os dados pelas chave, uma por vez.

# Algoritmo Map Reduce

Detalhes de funcionamento:



# Algoritmo Map Reduce

Detalhes de funcionamento:

Tarefas *Map*:

- Um bloco no DFS possui vários 'membros' que serão processados. Cada membro só pertence a apenas 1 bloco;
- A estrutura chave valor é importante pois permite a execução de várias tarefas *Map* em paralelo;
- A função *Map* é responsável por converter os dados para a estrutura chave valor;
- Chaves não são 'chave', no sentido estrito, não precisam ser únicas;

# Algoritmo Map Reduce

Detalhes de funcionamento:

Tarefas *Map*:

- Exemplo clássico - Contar o número de palavras em uma coleção de documentos:
  - Cada tarefa *Map* lê um documento ou vários documentos;
  - Cada palavra será considerada uma chave:  $w_1, w_2, \dots, w_n$ ;
  - A seguinte estrutura chave valor será criada:
    - $\langle w_1, 1 \rangle$
    - $\langle w_2, 1 \rangle$
    - ..
    - $\langle w_n, 1 \rangle$

**Sugestões para melhorar a função *Map* ???**



# Algoritmo Map Reduce

Detalhes de funcionamento:

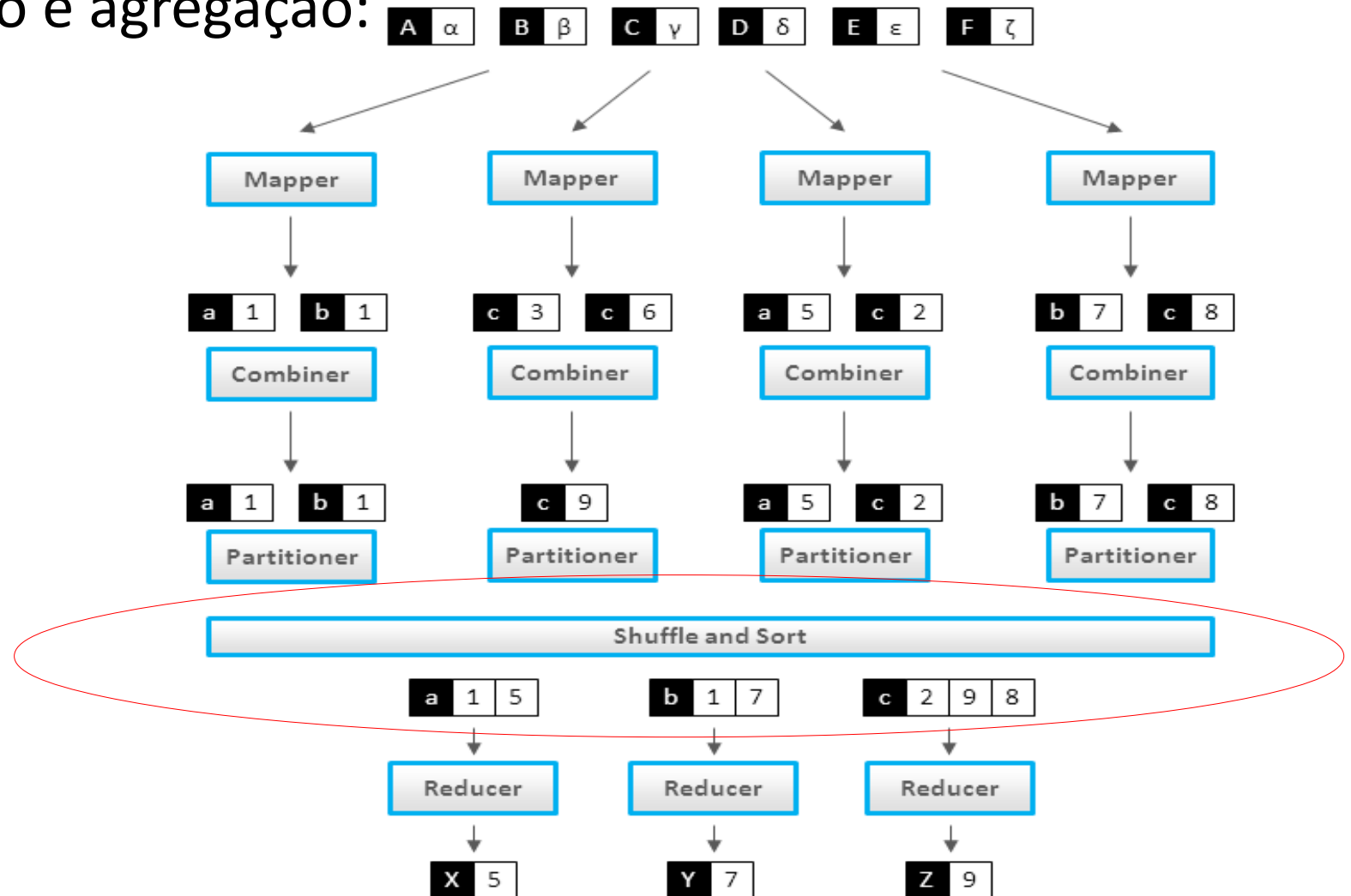
Agrupamento e agregação:

- São realizados, sempre da mesma forma, independente do que as funções *Map* e *Reduce* façam:
- O “*Name node*” controla este processo:
  - O número de tarefas *reduce* já são conhecidos:  $r$  (pode ser previamente determinada);
  - Cria um função *hash* (0 até  $r-1$ ) que é aplicada nas chaves;
  - Cada chave gerada pela tarefa Map é então gravada em um dos  $r$  arquivos locais;
  - Após todas as tarefas *map* serem finalizadas o “*master node*” faz um *merge* dos arquivos e que são então destinadas para as tarefas de *reduce*.

# Algoritmo Map Reduce

Detalhes de funcionamento:

Agrupamento e agregação:



# Algoritmo Map Reduce

Detalhes de funcionamento:

Tarefas *Reduce*:

- Para cada chave  $k$ , a tarefa *reduce* recebe um conjunto de pares na forma  $(k, [v_1, v_2, \dots, v_n])$  oriundos de várias tarefas *map* na forma  $(k, v_1)(k, v_2) \dots (k, v_n)$ ;
- A tarefa *reduce* deve combinar os valores de alguma forma;

# Algoritmo Map Reduce

Detalhes de funcionamento:

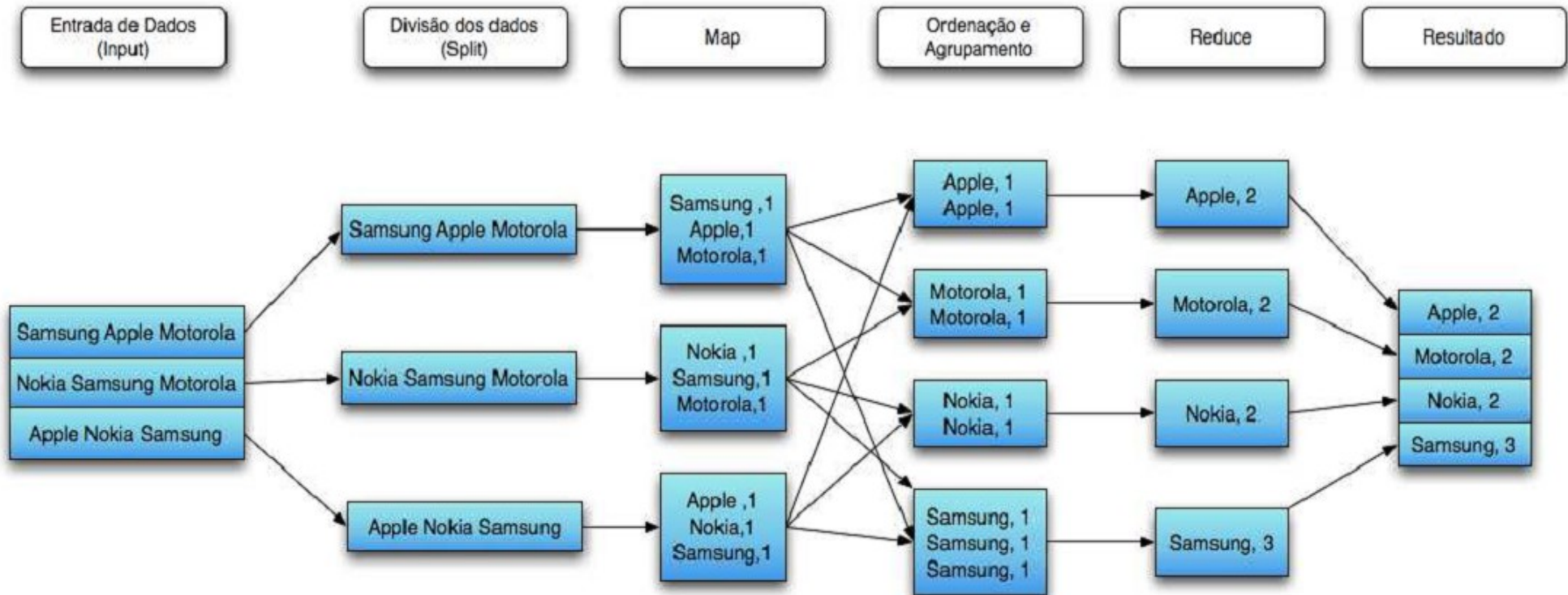
Tarefas *Reduce*:

- Exemplo clássico - Contar o número de palavras em um coleção de documentos:
  - A função *reduce* apenas faz a soma dos valores para cada uma das chaves  $w_n$
  - Desta forma a saída da função *reduce* será um conjunto de chaves-valor na forma  $(w,m)$ ;
  - Em que  $m$  é o total de vezes que a palavra  $w$  aparece em todos os documentos;

# Algoritmo Map Reduce

Detalhes de funcionamento:

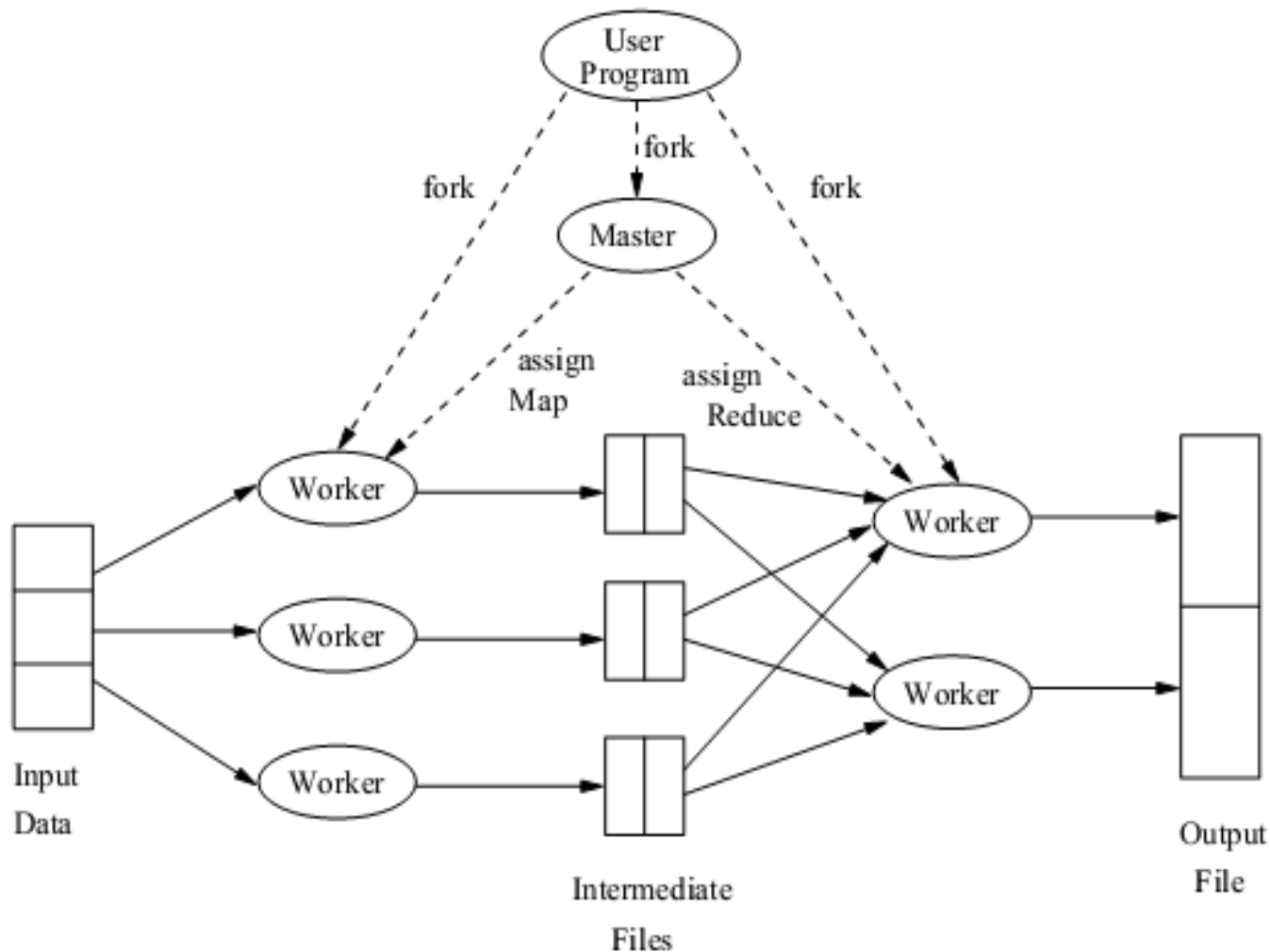
Exemplo clássico:



# Algoritmo Map Reduce

Detalhes de funcionamento:

Outras considerações: processos, arquivos e tarefas



# Algoritmo Map Reduce

Detalhes de funcionamento:

Outras considerações:

- Revisão: processos, arquivos e tarefas
  - O '*Name node*' é responsável pela criação das tarefas de Map e Reduce nos '*data nodes*' ;
  - É ideal que se crie um tarefa *map* para cada bloco;
  - O número de tarefas *reduce* deve ser mais reduzido, a intenção é evitar a explosão do número de arquivos gerados pela tarefa *map* ( um arquivo para cada *reduce*);
  - O '*name node*' acompanha a execução das tarefas (em espera, executando, completo); Quando um processo termina ele comunica para o '*name node*';

# Algoritmo Map Reduce

Detalhes de funcionamento:

Outras considerações:

- Revisão: processos, arquivos e tarefas
  - Cada tarefa *map* processa vários blocos de vários arquivos;
  - Os arquivos gerados pelo *map* são gravados localmente e o '*name node*' armazena todos estes dados: nomes dos arquivos e tamanho;
  - Uma tarefa *reduce* recebe todos os arquivos intermediários para o processamento da função *reduce*;



# Algoritmo Map Reduce

## Resumo Python:

### Variáveis e Substituição:

```
lista = [ 1, 2, "texto", 3.5 ]
print lista[ 0 ]      # imprime 1
print lista[ 1 : 2 ]  # imprime [ 2, "texto" ]
print lista[ : -1 ]   # imprime [ 1, 2, "texto" ]
lista += [ "novo" ]
print lista           # imprime [ 1, 2, "texto", 3.5, "novo" ]

tupla = ( 1, 2, "texto", 3.5 ) # Elementos não podem ser alterados!
print tupla[ 0 ]      # imprime 1
print tupla[ 1 : 2 ]  # imprime ( 2, "texto" )
print tupla[ : -1 ]   # imprime ( 1, 2, "texto" )
tupla += ( "novo", )
print tupla           # imprime ( 1, 2, "texto", 3.5, "novo" )

dicionario = { "chave": "valor", "c2": "v2" }
print dicionario[ "chave" ] # imprime valor

newstring1 = "string='%s' int='%d' float='%03.2f'" % ( "txt", 12, 4.56 )
newstring2 = "chave=(chave)s c2=(c2)s" % dicionario
newstring3 = "chave=%s c2=%s" % ( dicionario[ "chave" ],
                                dicionario[ "c2" ] )
```

### Controle de Fluxo e Laços:

```
if a > b and a < c:
    print "a entre b e c"
elif a > c:
    print "a maior que c"
else:
    print "a menor ou igual a b ou igual a c"

for elemento in lista:
    print "elemento: %s" % elemento

coordenadas = [ ( 0, 0, 0 ), ( 1, 0, 0 ), ( 0, 1, 0 ), ( 0, 0, 1 ) ]
for x, y, z in coordenadas:
    print "Ponto: x=%d, y=%d, z=%d" % ( x, y, z )

loop = 1
while loop:
    resultado = faca_acao()
    if resultado < 0:
        break # Para o laço
    else resultado > 0:
        continue # Volta para o começo do laço
    print "teste"
```

### Funções:

```
def funcao( p1, p2="Valor Padrao" ):
    print "p1: '%s' p2: '%s'" % ( p1, p2 )

def f_param_variaveis( p1, *args ):
    print "p1: '%s'" % p1
    for arg in args:
        print "    arg: %s" % arg

def f_param_nome_variaveis( p1, **args ):
    print "p1: '%s'" % p1
    for p_name, p_value in args:
        print "    arg: %s=%s" % ( p_name, p_value )
```

### Classes:

```
class A:
    atributo = 1
    __privado = 123
    def __init__( self, valor ):
        self.atributo = valor
        self.__metodo_privado()
    def __metodo_privado( self ):
        print "chamando metodo privado"

class B:
    atributo = 2
    def __init__( self ):
        self.novo_atributo = 2

class C( A, B ):
    def __init__( self ):
        B.__init__( self )

class D( B, A ):
    def __init__( self ):
        B.__init__( self )

a = A( 1 )
b = B()
c = C()
d = D()

print a.atributo      # imprime 1
print b.atributo      # imprime 2
print c.atributo      # imprime 1: herança múlt. (A,B) A sobrepõe-se a B
print d.atributo      # imprime 2: herança múlt. (B,A) B sobrepõe-se a A
```

### Módulos e Espaço de Nomes:

```
import urllib
url = "http://www.unicamp.br/" + urllib.quote( "index.html" )
conteudo = urllib.urlopen( url ).read()

# importa símbolos para espaço de nomes atual:
from urllib import *
url = "http://www.unicamp.br/" + quote( "index.html" )
conteudo = urlopen( url ).read()
```

# Algoritmo Map Reduce

## Exemplo - Mincemeat:

```
#!/usr/bin/env python
import mincemeat

data = ["Humpty Dumpty sat on a wall",
        "Humpty Dumpty had a great fall",
        "All the King's horses and all the King's men",
        "Couldn't put Humpty together again",
        ]

def mapfn(k, v):
    for w in v.split():
        yield w, 1

def reducefn(k, vs):
    result = 0
    for v in vs:
        result += v
    return result

s = mincemeat.Server()

# The data source can be any dictionary-like object
s.datasources = dict(enumerate(data))
s.mapfn = mapfn
s.reducefn = reducefn

results = s.run_server(password="changeme")
print results
```

# Algoritmo Map Reduce

Exemplo – Mincemeat:

- Servidor – '*Name node*':

```
python example.py
```

- Clientes – '*data node*':

```
python mincemeat.py -p changeme [server address]
```

- Resultado:

```
{'a': 2, 'on': 1, 'great': 1, 'Humpty': 3, 'again': 1, 'wall': 1, 'Dumpty': 2, 'men': 1, 'had': 1, 'all': 1, 'together':
```

# Algoritmo Map Reduce

Tipo de processamentos:

- *Map reduce* não é uma solução para qualquer tipo de problema. Ex. de exceção: site de comércio eletrônico;
- Pode ser utilizado:
  - Encontrar palavras chaves;
  - Operações que envolvam multiplicação de matrizes;
  - Operações de álgebra relacional:
    - Seleção -  $\sigma$
    - Projeção -  $\pi$
    - União, interseção e diferença
    - Natural *join* -  $\bowtie$
    - Agrupamentos e agregações

# Algoritmo Map Reduce

Natural Join – SQL:

- Suponha as duas relações  $R(A,B)$  e  $S(B,C)$

Função *Map*:

- Para cada tupla  $a, b$  de  $R$  produza um membro chave valor  $(b, (R, a))$ ;
- Para cada tupla de  $b, c$  de  $S$  produza um membro chave valor  $(b, (S, c))$ ;

- Função *Reduce*:

- Cada chave  $b$  deve ser associada com ambos os itens  $(R, a)$  e  $(S, c)$ ;
- A saída para a chave  $b$  deve ser  $(b, [(a_1, b, c_1), (a_2, b, c_2), \dots])$

# Algoritmo Map Reduce

- Exercícios práticos
  - Exercício de contagem de palavras para textos
  - Exercício de implementação de natural join e agrupamentos;
- Trabalho prático
  - Palavras chaves por autores;

NoSQL MongoDB MapReduce Cassandra  
Python Processamento  
BerkeleyDB Paralelismo DFS  
chave-Valor Documentos Colunar

## Paradigmas de computação distribuída e bancos de dados

---

Consistência em bancos de dados  
não relacionais

# Consistência em bancos de dados não relacionais

- Consistência estrita
  - Todas as operações de leitura devem retornar dados da última operação de escrita completa;
  - Operações de leitura e escrita em um único nó;
  - Utilização de protocolos de transação distribuída;
  - De acordo com o teorema CAP, não pode ser conseguida em conjunto com disponibilidade e tolerância a partição;



# Consistência em bancos de dados não relacionais

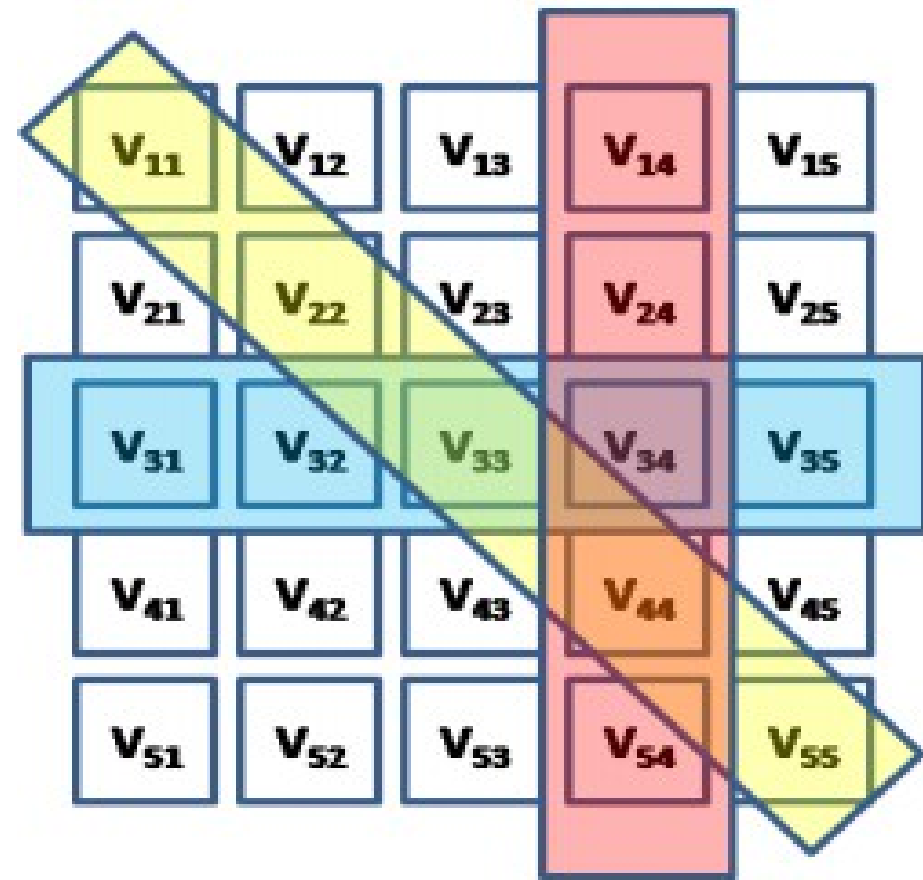
- Consistência eventual
  - As operações de leitura irão, eventualmente, ler dados da última operação de escrita;
  - Pode haver leituras inconsistentes pois o sistema geral esta em atualização;
  - Em um *cluster* uma leitura pode ser feita a partir de uma réplica que ainda não foi atualizada, pois o último processo de escrita aconteceu em outra réplica do dado;
  - Em alguns sistemas este tempo de atualização das réplicas pode ser 500 milisegundos;

# Consistência em bancos de dados não relacionais

- Consistência eventual
  - Se bancos de dados são distribuídos (vários nós), há a chance de dados serem lidos e alterados em qualquer nó;
  - Consistência estrita não é possível neste cenário;
  - Desta forma existe a necessidade de controlar modificações e versões concorrentes para quais o estado final do banco de dados vai convergir;
- Tratamento e versionamento: ambientes distribuídos
  - Timestamp: exige a necessidade de um relógio sincronizado em todo *cluster*;
  - Vetores de relógios: muito usado, mas exige alguma forma de resolver conflitos;

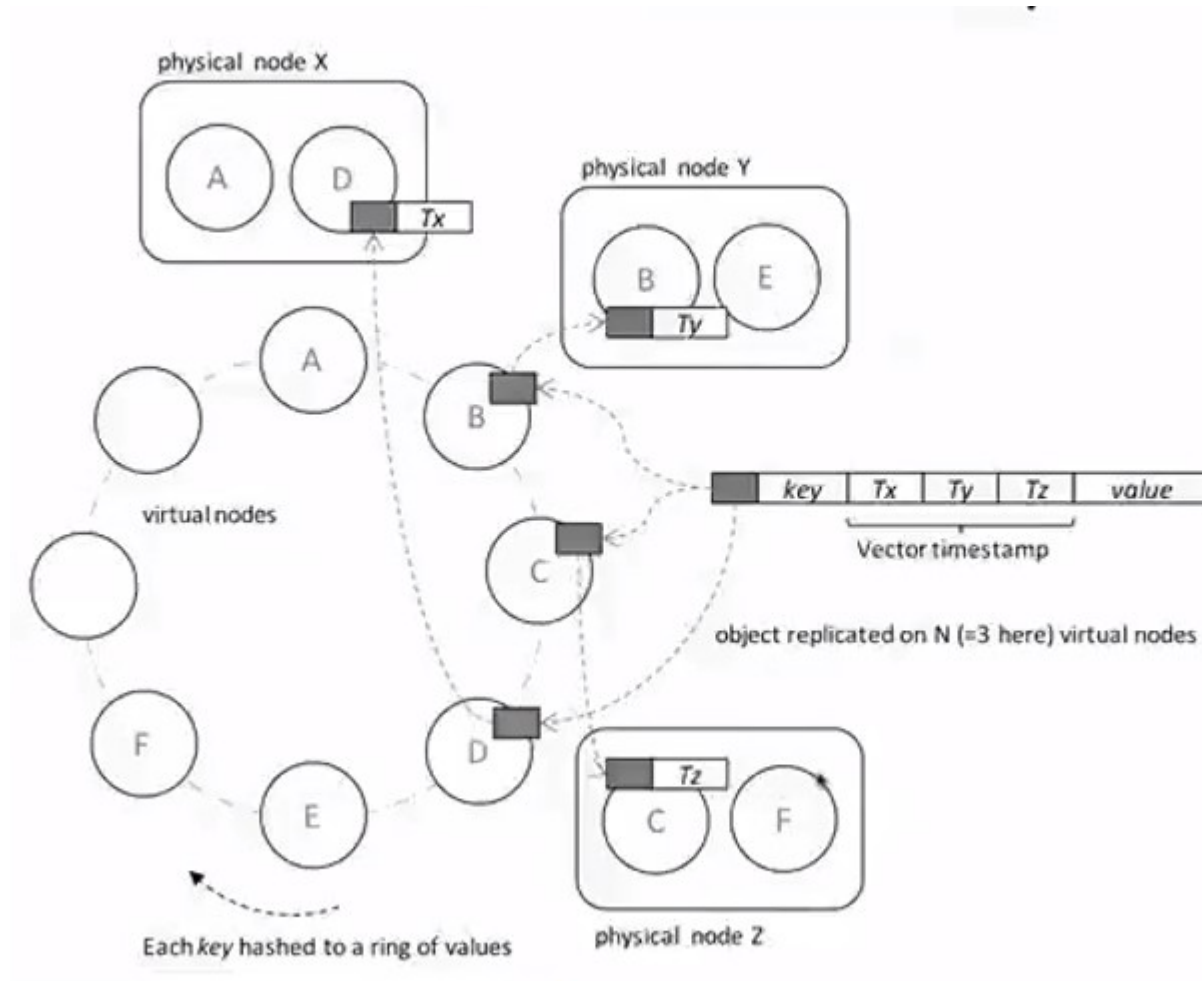
# Consistência em bancos de dados não relacionais

- Vetores de relógios
  - Item  $i$  da cópia local – **diagonal**;
  - O que a replica sabe sobre a atualização das demais replicas – **linhas** da matriz;
  - O que todos sabem sobre uma determinada réplica – **colunas** da matriz ;



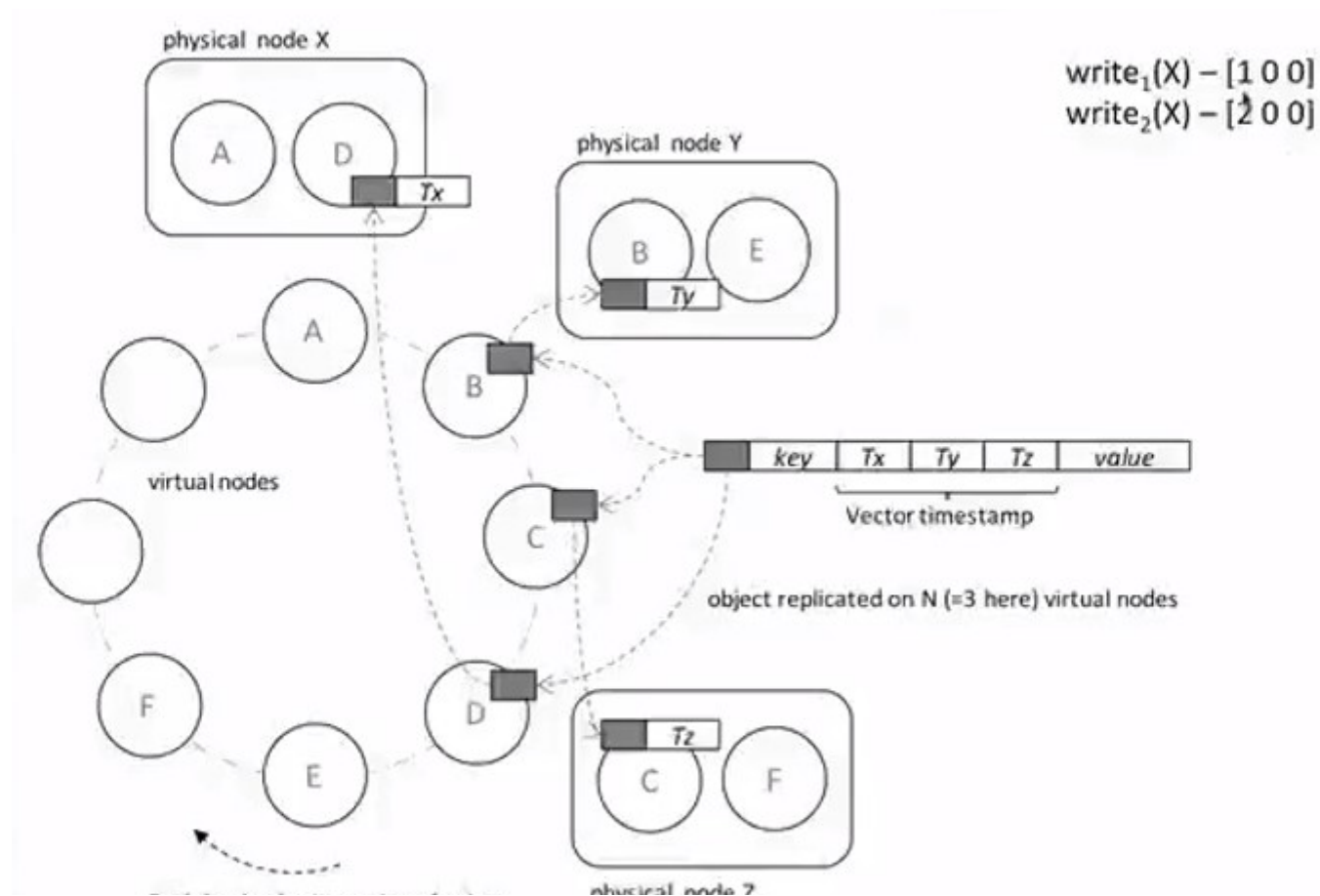
# Consistência em bancos de dados não relacionais

- Vetores de relógios



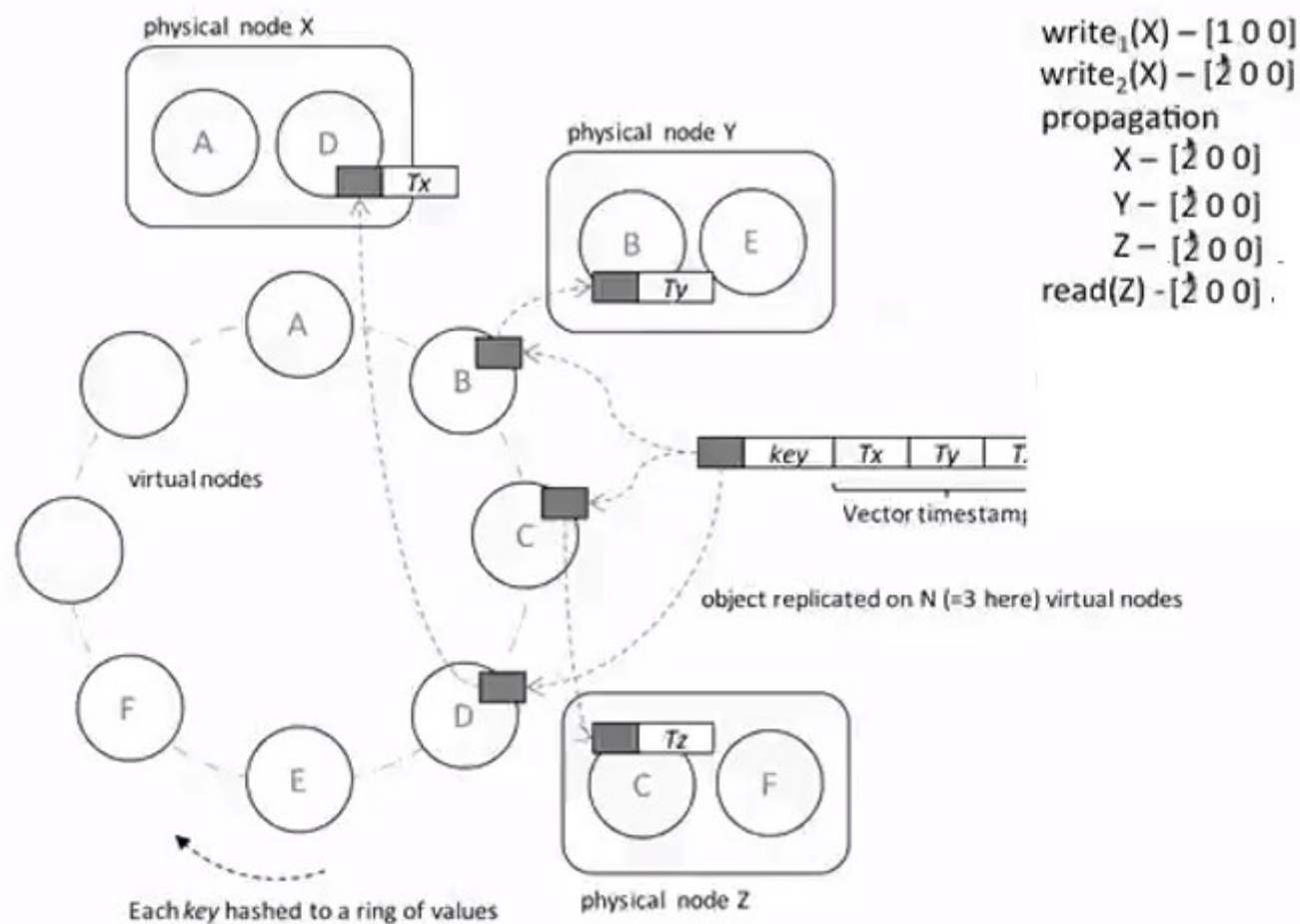
# Consistência em bancos de dados não relacionais

- Vetores de relógios



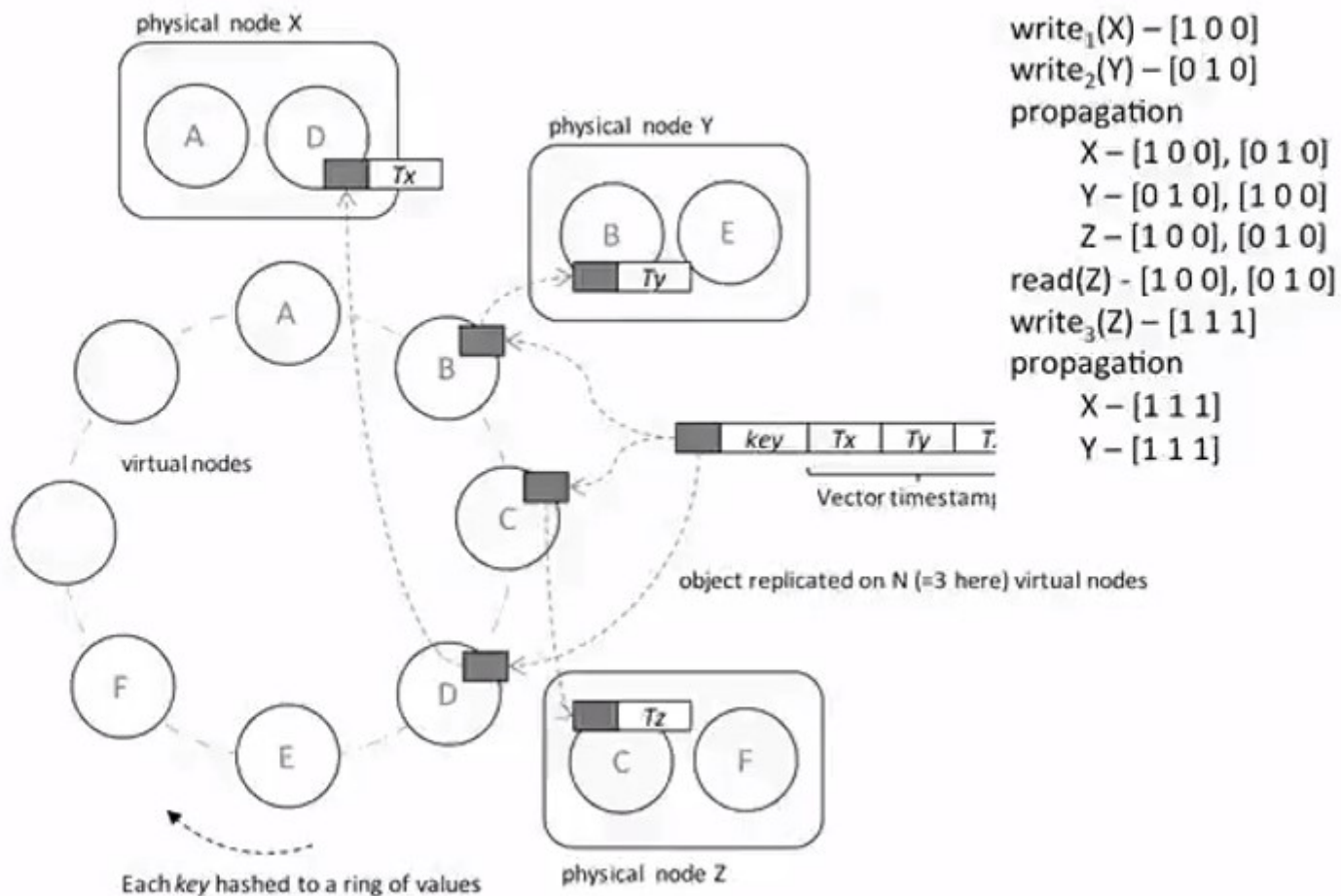
# Consistência em bancos de dados não relacionais

- Vetores de relógios



# Consistência em bancos de dados não relacionais

- Vetores de relógios



# Consistência em bancos de dados não relacionais

- Vetores de relógios
  - As resoluções de conflitos, assim como os vetores de relógios, possuem uma teoria mais extensa;
  - Outras tecnologias NoSQL utilizam outras abordagens;
  - A propagação e resolução dos conflitos envolve:
    - Propagação de dados;
    - Propagação de operações;
    - Vetores de relógios também podem estar nos clientes;
    - Sistemas de filas para replicar operações – negociação entre as replicas sobre a ordem das alterações;



**Consistência eventual - Alexandre Porcelli – DevInVale -2011**



# Atividade

Atividade:

Trabalho prático – *Map Reduce*;

Reflexão

- Quais são as principais características dos **sistemas de arquivos distribuídos**?
- Qual a proposta do paradigmas de processamento ***Map reduce***? Podem ser aplicados em qualquer cenário de computação?
- O que as funções ***Map*** e ***reduce*** fazem especificamente?
- Que tipos de problemas existem no quesito **consistência**, considerando: um banco de dados distribuído, orientado a tolerância a partição e disponibilidade.
- Em que cenários a **consistência eventual** não seria um problema?