**Sistemas de Computação em Cloud**

### First generation batch processing: Map-reduce

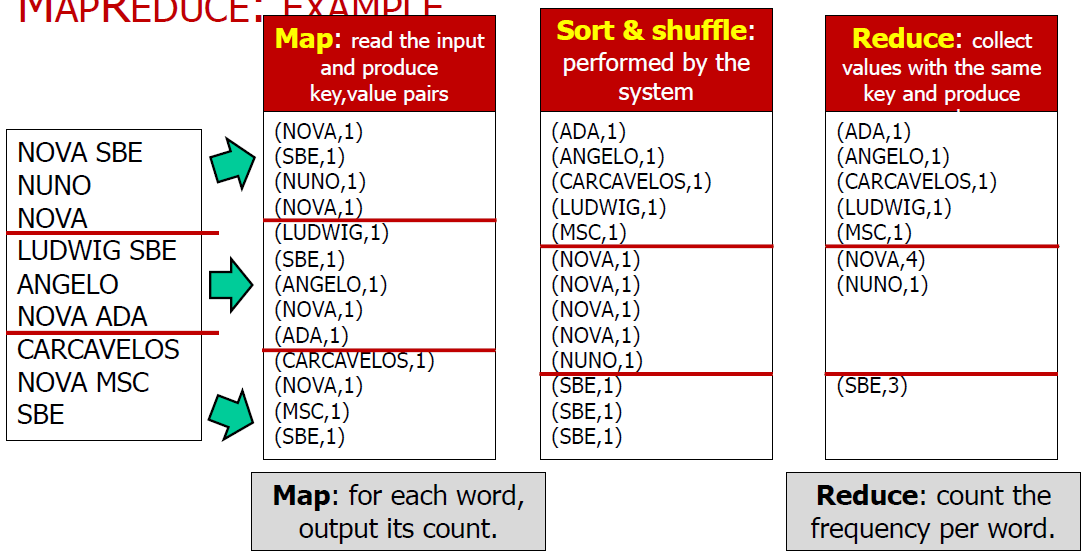
Permite expressar cálculos simples que estávamos tentando realizar, mas esconde os detalhes confusos de:

* paralelização
* tolerância a falhas,
* distribuição de dados
* balanceamento de carga em uma biblioteca

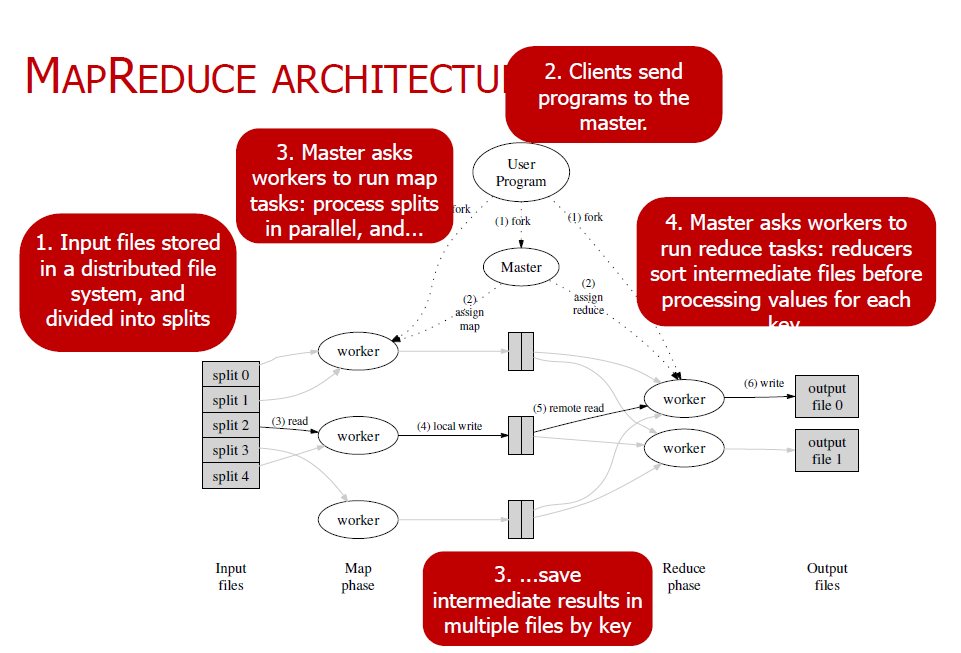
Um modelo de programação e uma implementação associada para processar grandes conjuntos de dados

## OverView

1. Lê sequêncialmente muita informação
2. Map phase – extrai a info. importante
3. Agrupa por chave (Sort and Shuffle do output da Map phase – por frequência da palavra)
4. Reduce phase - Aggregate, summarize, filter or transform
5. Dá o resultado



# Arquitetura



## Master Node

O **Master node** coordena a execução:

* Task status: (idle, in-progress, completed)
* Idle Tasks: as tarefas ociosas são agendadas à medida que os trabalhadores ficam disponíveis
* Quando uma tarefa de mapa é concluída, ela envia ao mestre a localização e os tamanhos de seus arquivos intermediários, um para cada redutor
* O master envia essas informações para os redutores
* O mestre pinga os trabalhadores periodicamente para detectar falhas

## Worker

O nó do trabalhador executa tarefas de mapeamento ou redução, conforme solicitado pelo coordenador.

## Handling Faults

**Map worker failure**

* Após a detecção da falha de um trabalhador, as tarefas do mapa são reiniciadas em diferentes trabalhadores

**Reduce worker failure**

* A tarefa de redução é reiniciada em outro trabalhador

**Stragglers (slow workers)**

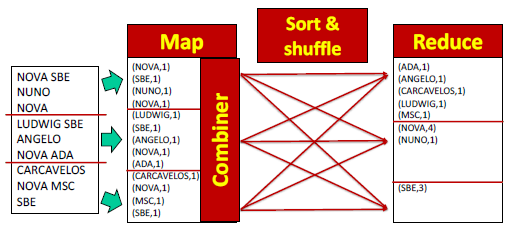
* Se uma tarefa estiver demorando muito para ser concluída, ela será iniciada em outro trabalhador. E usado o primeiro resultado.

**Falha mestre**

* A tarefa MapReduce é abortada e o cliente é notificado

# Map reduce Improvement – Combiner

Combiner permite pré-agregar valores no mapper. Normalmente o mesmo que a função de redução



# Desvantagens

O MapReduce requer cálculos complexos para serem divididos em sucessivos MapReduce jobs

Apresentam alta latência devido a:

* + precisa ler e gravar arquivos
  + replicação do sistema de arquivos fundamental (para gravações)
  + um trabalho deve terminar antes que o próximo possa ser iniciado

O Apache Spark aborda essas limitações.

### Second generation batch processing: Spark

Apache Spark fornece:

* processamento distribuído na memória
* tolerante a falhas

Os programas Spark tem:

* + múltiplas transformações de dados encadeadas
  + modelo de programação funcional de alto nível
  + define uma estrutura de dados de coleta distribuída: **Resilient Distributed Dataset (RDD) –** registos de dados que residem em vários nós.

Cada conjunto de dados no **Spark Resilient Distributed Dataset** é dividido em partições no cluster podendo ser operado em paralelo, em diferentes nós do cluster. Ele também tem o potencial de se recuperar de ocorrências de falhas no sistema automaticamente

# Data Models e API’s

## RDDs

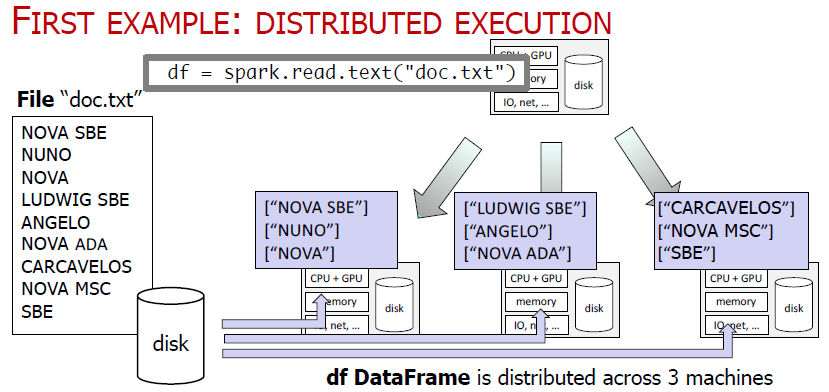
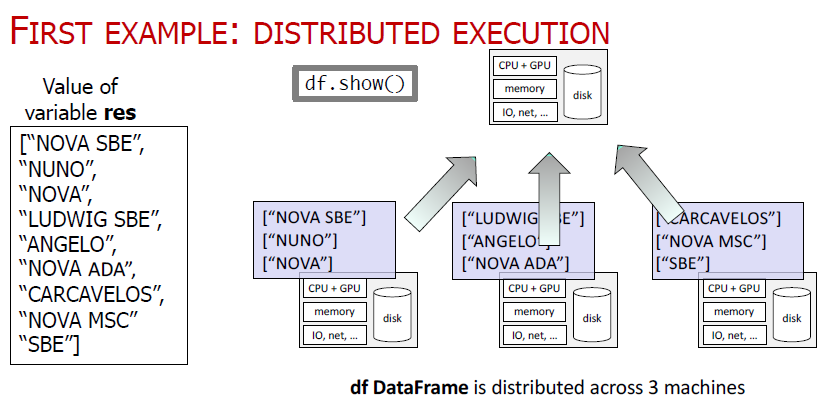
* + RDD – tuplos de dados imutáveis
  + particionado em vários nós;
  + após uma falha os RDDs podem ser automáticamente recriados e de forma eficientemente, a partir das dependências.

## Quadros de dados do Spark

**DataFrames -** coleções distribuídas de dados agrupadas em colunas com um nome. Podem ser vistas como RDD’s com um esquema que nomeia os campos dos tuplos fundamentais.

#### Spark SQL

* + - Programas com SQL/DataFrames 🡪 convertidos em programas Spark.
    - Programas Spark são otimizados para serem executados com eficiência.
    - Spark SQL usa técnicas de sistemas de BD.
    - Programas Spark são bons para: processar gráficos e aprendizado de máquina.

# Programming model

**Programas Spark Dataframe** descrevem o fluxo de transformações que cria uma DataFrame a partir de outra.

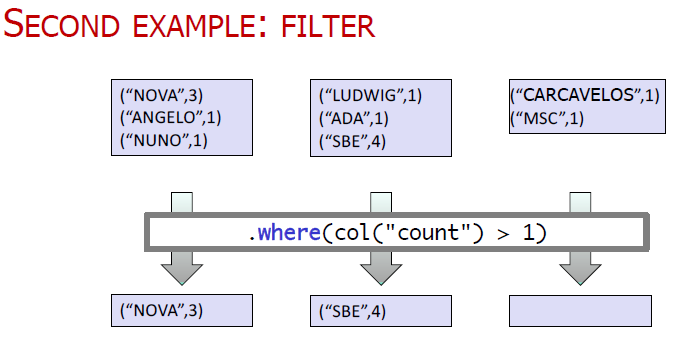
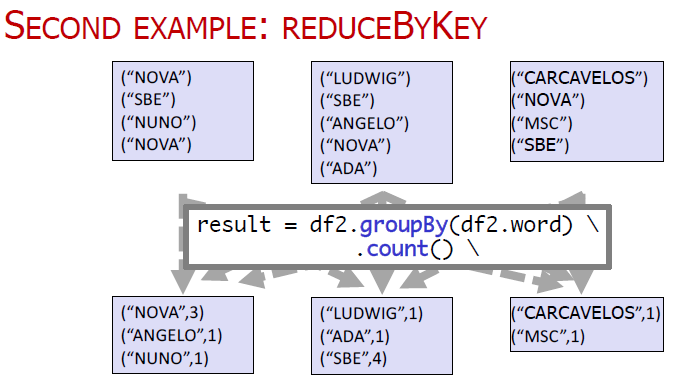
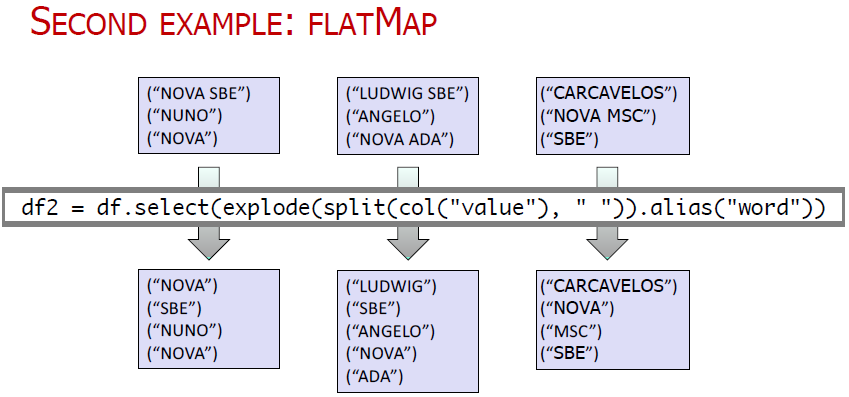
**Programas Spark** codificam as dependências entre os vários DataFrames (e RDDs) - gráfico de linhagem.

Wide-Dependencies

Produzidas quando uma partição RDD depende de várias partições armazenadas em nós diferentes (alto custo de bandwidth no groupByKey)

Narrow-dependencies

Produzidas quando uma partição RDD depende de dados que estão no mesmo nó



# Programming and execution model

Programas DataFrame 🡪 convertidos 🡪 programas RDD, que envolvem:

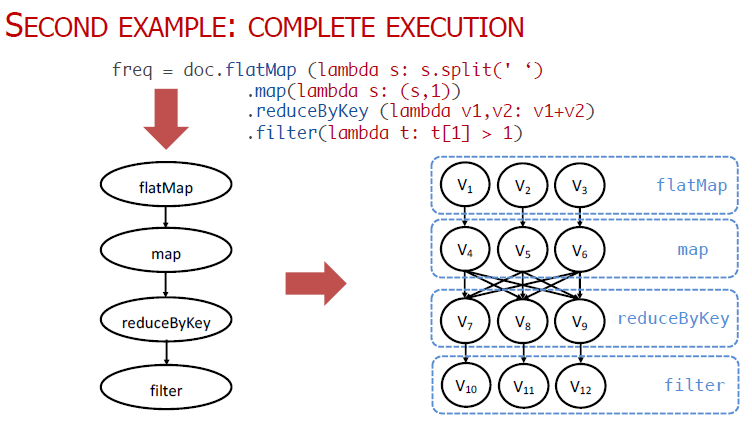
* **Transformações**: RDD -> RDD
* **Ações**: RDD -> Resultado (disponível diretamente para o aplicativo cliente)

O **RDD** fornece APIs de baixo nível para processamento de dados distribuídos. Por outro lado, **DataFrame** fornece APIs de alto nível que suportam métodos SQL.

## Execução

* Aplica todas as partições de um RDD em paralelo.

**Melhor desempenho** quando o resultado de uma partição RDD não requer dados de partições RDD de entrada localizadas em nós diferentes (ou seja, evita embaralhamentos)



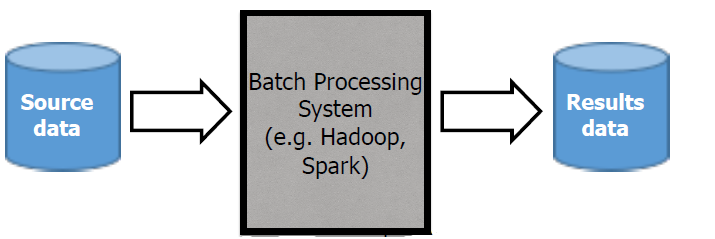
# Fault tolerance

Sparks lida com falhas de nó recalculando partições perdidas, usando informações de linhagem. Otimizado por RDDs intermédias persistentes.

No exemplo, se **V9 for persistido**, muitos recálculos serão salvos.

### Stream processing (Big data / batch processing)

**Objetivo:** Executar computação sobre dados e produzir resultado

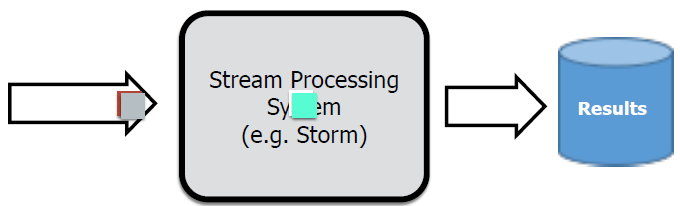


Exemplo: Produzir informação sobre o trânsito com base na informação recolhida dos telemóveis dos users

# Processing models

## Continuous

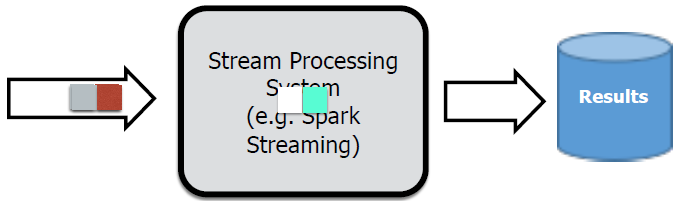
* Cada tuplo é processada à medida que chega
* O sistema de processamento pode manter o estado para executar **window computation** e **incremental computation**



## Mini- batches

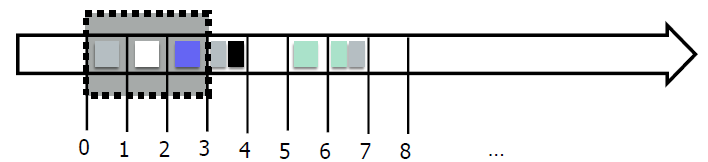
Tuplas recebidas para cada Xms agrupadas em um mini-batches

* Processar mini-batches
* O sistema de processamento pode manter o estado para executar **window computation** e **incremental computation**



# Windowing

Calcula-se resultados com base em dados de uma window, mas calcular os resultados mais frequentemente do que o intervalo de tempo (processar dados dos últimos 3 minutos, mas produzir resultados a cada minuto)



### Virtualization

Criação de recursos computacionais simulados (**recuros** - rede, armazenamento, CPU+memória).

## Vantagens

* + Uma VM fornece a abstração de uma máquina para cada cliente numa máquina física.
  + Permite compartilhar os mesmos recursos físicos entre vários usuários.
  + Recursos físicos e virtuais estão desacopulados.

## VMM

O VMM fornece a ilusão de uma máquina idêntica à real que controla o hardware. O software é executado em uma VM em completo isolamento de outras VMs (apenas com reduções na velocidade pequenas)

# Defenições

## Máquina Virtual (VM)

Simula a máquina real com um hardware virtual criado e gerenciado pelo VMM

* Várias instâncias de VM em execução podem:
  + comportar-se como máquinas distintas, isoladas
  + executar seus aplicativos sem modificações e de forma eficientemente

Virtual Machine Monitor (VMM)

Fornece as abstrações de hardware virtual para a VM

## Hypervisor

É um SO executa os processos (VM/VMM),

* Um par (VM+VMM) constitui um “processo” de hypervisor

## Language VM

Fornece um ambiente com um conjunto de instruções específico e bibliotecas padrão para aceder os recursos do computador.

* Executa como um aplicativo em um SO e os programas são compilados para o código VM do idioma e executados no idioma VM.

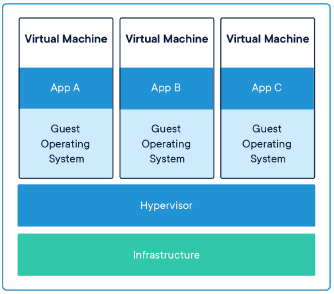
# Virtualização

A virtualização permite executar um SO dentro de outro

* Host OS/Hypervisor 🡪 rodando na máquina real
* SO convidado 🡪 corre dentro do SO host

O VMM (ou hypervisor) deve fornecer:

* Segurança: a VM não deve comprometer o VMM, que deve ter controle total dos recursos (reais e virtuais).
* Fidelidade:durante a execução, o comportamento do software na VM deve ser idêntico ao comportamento do hardware real.
* Eficiência:o VMM deve se interpor o mínimo possível enquanto o VM está executando seu código.

**Como o VMM se protege de VMs?**

O emulador (VMM) pode manter o estado:

* do processador;
* da memória,
* do IO

e interpretar cada instrução da VM, atualizando o estado do processador/memória/IO real de acordo (seguro, mas muito lento). Para ser rápido, é necessário que o software da VM rode no processador fisico todo, mas sem comprometer o sistema.

## VM garante

1. Isolamento de memória entre processos;
2. Operações privilegiadas - permitem que um processo acessa-se recursos do computador mas no modo supervisor (código do SO é o único que é executado no modo supervisor)

**Os VMMs podem usar a mesma abordagem?**

* Não se houver apenas dois modos (S & U)
* Se SO convidado da VM for executado em modo privilegiado, isso poderá comprometer o VMM, pois teria acesso a todas as instruções;
* Se o sistema operacional convidado da VM for executado no modo de usuário, ele não poderá ser executado corretamente, pois não poderá executar as instruções privilegiadas necessárias.

## VMM garante

1. O OS convidado da VM precisa pensar que está sendo executado no modo supervisor precisa ver o sinalizador S(supervisor);
2. O OS convidado da VM precisa ser capaz de emitir instruções privilegiadas, mas sua execução precisa ser controlada pelo VMM.

# Implementação - Virtualização com Trap-and-emulate (friendlly)

Interceptação de instruções privilegiadas se executadas no modo de Utilizador

**Operações sensíveis** - comportam-se de maneira diferente de quando executadas nos modos U e S

1. A VM executa o código do cliente diretamente na CPU (sem penalidade de desempenho)
2. O OS convidado é executado no modo de U (mas os sinalizadores vCPU indicam o modo S - portanto, o OS convidado pensa que está sendo executado no modo de usuário)
3. Quando o OS convidado executa uma instrução privilegiada, um trap altera a execução para VMM (em um trap, a execução do OS convidado é interrompida e uma função do VMM é chamada).
4. O VMM interpreta (emula) a instrução e retorna a execução para o sistema operacional convidado (no modo de usuário).

# Implementação - Virtualização com Binary translation (unfriendlly)

Instruções sensíveis não são privilegiadas 🡪 PUSHF, POPF.

**Problema:** se o SO convidado for executado no modo U (com o sinalizador vCPU S mode), o efeito da execução da operação não será correto.

**Ideia:** substituir a instrução sensitiva por um trap (forçando execução controlada no VMM).

* A VM executa o código do usuário diretamente na CPU (sem penalidade de desempenho).
* O OS convidado é executado no modo U (mas os sinalizadores vCPU indicam o modo S).
* O código binário do OS convidado é alimentado a um tradutor que emite código binário alterado para instruções confidenciais
* Instruções privilegiadas são executadas como trap-and-emulate.

#### Desvantagens

* Lentidão no código do OS convidado – necessidade de traduzir o código;
* Várias instruções podem precisar ser capturadas em cada função do OS convidado;

# Implementação - Virtualização com Paravirtualização (unfriendlly)

**Problema:** mesmo que a tradução de código possa ser rápida, ter várias instruções de cada função do SO convidado bloqueadas retarda a execução.

**Ideia:** Uma única trap seria necessária para chamar a função VMM. Além disso, a função VMM pode acessar os recursos diretamente (em vez de acessar uma versão virtualizada).

* O código-fonte do SO convidado é alterado para que as partes que usam instruções confidenciais não privilegiadas e privilegiadas sejam substituídas por chamadas de hipervisor que executam a mesma funcionalidade (por exemplo, uma função de OS convidado que acederia ao controlador de memória virtual é substituída por uma chamada VMM/hypervisor que acede ao controlador de memória virtual diretamente)
* A VM executa o código do usuário diretamente na CPU (sem penalidade de desempenho).
* O SO convidado é executado no modo U (mas os sinalizadores vCPU indicam o modo S).
* As chamadas do hipervisor são implementadas como traps.

#### Vantagens

Desempenho dentro de alguns por cento do caso não virtualizado.

#### Desvantagens

Requer alterações no SO convidado: ok quando a fonte estiver disponível (por exemplo, Linux).

# Hoje em dia

A **paravirtualização** resolveu o problema de desempenho quando a tradução binária foi usada pois não é necessário com o suporte de hardware para virtualização.

# Virtualização de CPU’s

Virtualização de back-end

A pilha do driver gráfico é executada dentro da máquina virtual com o limite de virtualização entre a pilha e o hardware físico da GPU

#### GPU PCI passthrough technic

Permite uma VM tem acesso direto à GPU mapeando as suas regiões de memória no espaço de endereço da VM.

* **Vantagem:** Desempenho quase nativo paraGPUs
* **Problemas:** Sem multiplexação

Virtualização de front-end

A virtualização de front-end introduz um limite de virtualização em um nível relativamente alto na pilha e executa o driver gráfico no host/hipervisor

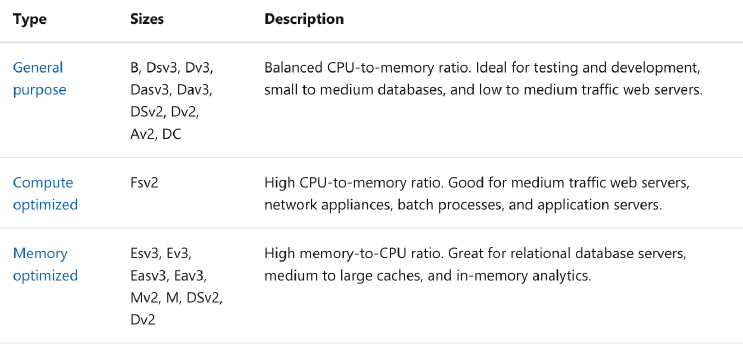
#### Technic

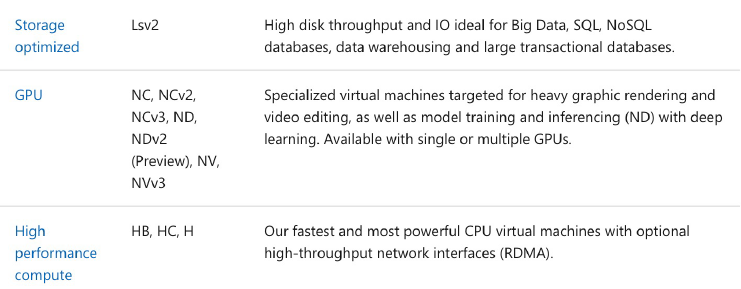
A GPU é controlada pelo host/hipervisor e oferece suporte à multiplexação. Na **emulação** de dispositivos, um GPU a sua emulação sintetiza as operações gráficas do host em resposta às ações dos drivers do dispositivo convidado. Isto permite que o GPU seja executado em uma máquina diferente.

### IaaS – Azure VM

* Suporta a criação de MV rodando Linux e Windows.
* Suporta a criação de VMs especialmente adaptadas para a execução de software específico
* Utilizador seleciona as VMs que precisa.

# Type of each VM





# VM and faults

Unplanned Hardware Maintenance Event

Ocorre quando o Azure prevê que o harware/componente associado a uma máquina está prestes a falhar. Para isto a plataforma emite um evento de manutenção de hardware não planejado (pode usar o **Live Migration** para migrar as VM’s do hardware com falha para uma máquina física saudável)

#### Live Migration

Pausa a VM 🡪 migra memória, arquivos abertos e conexões de rede 🡪 retoma a VM.

Pode ser feito enquanto a VM está em execução, parando apenas por um curto periodo no final para migrar o que foi alterado entretanto

## An Unexpected Downtime

Ocorre quando o harware/componente associado a uma máquina falha inesperadamente (falhas de rede local, falhas de disco local ou outras falhas no nível do rack). Quando detectadas, a cloud platform migra automaticamente a MV para uma máquina física saudável no mesmo datacenter.

Durante a recuperação, as MV’s sofrem um tempo de inatividade (reinicialização) e, em alguns casos, dá-se perda da unidade temporária. Contúdo:

* O SO anexado e os discos de dados são sempre preservados.
* As máquinas virtuais também podem sofrer inatividade no caso improvável de uma interrupção ou desastre que afete um datacenter inteiro ou até mesmo uma região inteira.

## Planned Maintenance events

Atualizações periódicas feitas na cloud platform fundamental para melhorar a confiabilidade, o desempenho e a segurança gerais da infraestrutura da plataforma.

As plataformas de nuvem tentam usar a **manutenção de preservação** **de VM** sempre que possivél e mantendo as VMs em execução (migrando-as para outras máquinas).

# Select the availability options

Cada **zona de disponibilidade** tem uma fonte de energia, rede e refrigeração distintas. Um aplicativo deve ter replicação da app e data em diferentes zonas para tolerar a perda de um datacenter.

#### Fault domains

Hardware fundamental que compartilha uma fonte de energia e um switch de rede comuns, semelhante a um rack em um datacenter local.

#### Update domains

Hardware fundamental que pode passar por manutenção ou ser reinicializado ao mesmo tempo.

## Scale set

O Azure permitem criar e gerenciar um grupo de VMs com balanceamento de carga, onde o número de instâncias de VM pode aumentar ou diminuir automaticamente em resposta à demanda ou a um cronograma definido. Para maximizar a disponibilidade e resiliência a interrupções, as VMs de um grupo podem ser implantadas em várias regiões.

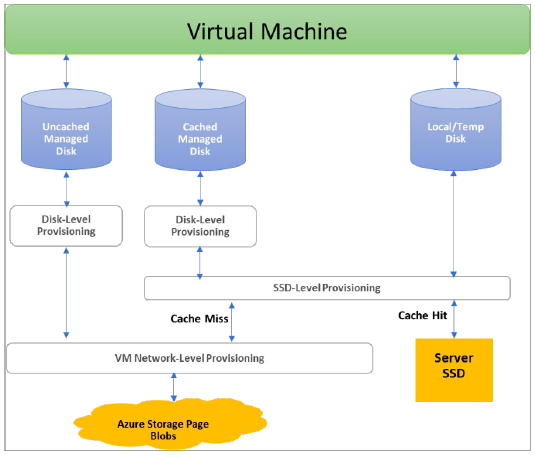
## Availability set

Um **availability set** é um agrupamento de VMs em um datacenter que fornece redundância e disponibilidade.

Com isto é possivél prevenir:

* falhas hardware
* interrupções de rede
* interrupções de energia

# Disks and network options

**Managed disk** é um disco rígido virtual armazenado como blobs de páginas.

## Tipos de disco

* disco Ultra,
* unidade de estado sólido (SSD) Premium,
* SSD padrão e unidade de disco rígido (HDD) padrão.

O sistema provisiona por disco IOPS, e largura de banda. Além disso, provisiona a rede para acesso remoto.

Localmente, o servidor implementa o provisionamento SSD para dados armazenados no SDD do servidor, incluindo disco com cache e discos locais e temporários.

# VM’s Vantagens vs. Desvantagens

#### Vantagens

**Eficiência -** uso eficiente de recursos e fornece isolamento.

**Flexibilidade -** recursos podem ser alocados conforme necessário.

**Restaurar e recuperar -** podem ser armazenadas como arquivo único que pode ser copiado noutra fonte.

**Liberdade do sistema operacional -** diferentes OSs convidados podem existir no mesmo hypervisor.

**Desempenho e movimento -** os hipervisores suportam movimentação de uma VM entre hosts em caso de degradação do desempenho na máq. host.

#### Desvantagens

**Sobrecarga de desempenho -** uma pilha de VM’s inclui: o SO convidado, o hipervisor e potencialmente o SO host

**Utilização eficiente de recursos -** o uso de vários SO’s no mesmo hipervisor duplica os recursos usados.

### Containers

Os contêineres fornecem:

* virtualização no nível do OS.
* namespace privado
* interface de rede e endereço IP
* sistemas de arquivos
* cria uma vez e corre em qualquer lugar
* portablidade entre máquinas
* gerenciamento simplificado de dependências

Ao contrário das VMs, os contêineres compartilham o kernel do sistema host com outros contêineres. O código do aplicativo precisa usar a API do sistema operacional host.

Nota: antes dos container existiu o **chroot** - especificava uma dir como root, e isto permitia um app aceder a ficheiros de outra.

# Kernel namespaces

Os namespaces do kernel dividem os recursos do kernel (processos, usuários, pilhas de rede, etc.) em uma instância por namespace.

Um processo visualiza apenas os recursos em seu namespace (existem atualmente 6 namespaces) mas partilha o kernel subjacente com outros contêineres.

#### Implementação

System calls:

* **clone()** - cria um novo processo e um novo namespace
* **unshare()** - cria um novo namespace e anexa o processo atual para ele.
* **setns()** - permite ingressar em um namespace existente.

#### Namespaces

**UTS -** nome de host e de domínio do namespace.

**Network -** cópia da pilha de rede, com suas próprias rotas, regras de firewall e dispositivos de rede. Cada Network Namespacetem seus próprios endereços IP (Nota: Veth (ethernet virtual) é usado como um canal entre dois namespaces)

**Mount -** Na criação, a árvore do sistema de arquivos é copiada para um novo espaço, com todas as montagens anteriores visíveis.

**PID -** Processos em diferentes namespaces PID podem ter o mesmo ID de processo.

**User -** Um processo terá um conjunto distinto de UIDs, GIDs e capacidades

**IPC -** Cada namespace obtém seus próprios objetos IPC e filas de mensagens POSIX.

# Cgroups

Mecanismo para aplicar limites de:

* recursos de hardware
* consumo de memória
* CPU dos contêineres
* acesso a um processo ou uma coleção

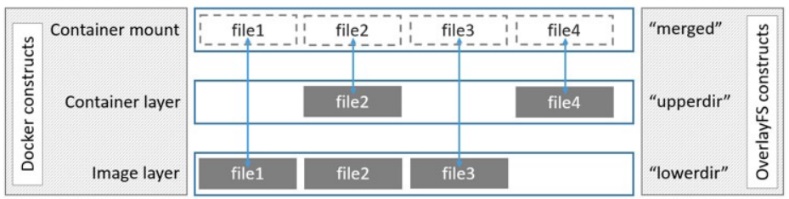
Nota: Um contêiner pode ser redimensionado simplesmente alterando os limites de seu cgroup correspondente.

#### Implementação

A implementação de cgroups requer mexer no kernel: na fase de inicialização, processo de criação e destruição. Todas as operações em cgroups são executadas usando operações em um VFS (sistema de arquivo virtual)

# Copy-on-write File system

**UnionFS** - sistema de arquivos copy-on-write que fornece:

* visão unificada do sistema de arquivos
* combina os sistemas de arquivos empilhados
* as gravações de um contêiner não afetam as leituras noutro
* vários contêineres podem partilhar dados comuns
* cada camada é armazenada apenas uma vez

# Docker

Construído sobre:

* namespaces do kernel
* cgroups
* unionFS

Cada contêiner:

* tem seu próprio conjunto de namespaces e Cgroups
* é isolado de outros (nem vê a lista de processos em outro) através dos namespaces
* Cgroups permitem que o administrador isole os recursos usados por cada contêiner e seus filhos

Outros

* executar daemon docker requer privilégios de root
* Docker fornece uma lista de permissões de recursos para usuários root num contêiner

#### Docker engine

**Daemon Docker -** gerencia objetos Docker como imagens, contêineres, redes e volumes. O cliente docker envia solicitações ao daemon docker.

**Registro do Docker -** armazena imagens do Docker.

**Imagem –** Somente para leitura com instruções para criar um contêiner, e pode ser criada a partir da especificação em um Dockerfile.

#### Docker networking

A **Bridge network** - conecta vários contêineres docker em execução no mesmo host docker. Uma rede pode ser criada usando o docker network create. O driver de rede de sobreposição cria uma rede distribuída entre vários hosts de daemon do Docker

#### Docker Storage

Os arquivos criados dentro de um contêiner são armazenados em uma camada de um contêiner que possa ser escrito e não persistente. O Docker tem duas opções para contêineres armazenarem arquivos na máquina host (para que os arquivos sejam persistidos mesmo após o container parar): volumes e montagens de ligação.



Os **Volumes** são armazenados em uma parte do sistema de arquivos host que é gerenciado pelo Docker. Os processos não Docker não devem modificar esta parte do sistema de arquivos. Os volumes são a melhor maneira de persistir dados no Docker.

As **Bind mounts** podem ser armazenadas em qualquer lugar no sistema host. Eles podem até ser arquivos ou diretórios importantes do sistema. Os processos não Docker no host Docker ou em um contêiner Docker podem modificá-los a qualquer momento.

As **tmpfs mounts** são armazenadas apenas na memória do sistema host e nunca são gravadas no sistema de arquivos do sistema host.

#### Docker compose

O Docker Compose permite definir e executar aplicativos Docker de vários contêineres.

# Docker Swarm

O Swarm é um cluster de instâncias do Docker, e consiste em vários hosts do Docker que são executados no modo Swarm e atuam como managers e trabalhadores.

**Funcionamento**

Ao criar um serviço, defenimos o número de réplicas, rede e recursos de armazenamento disponíveis para ele, portas que o serviço expõe ao mundo externo, e o Docker trabalha para manter esse estado. Por exemplo, se um nó de trabalho ficar indisponível, o Docker agendará as tarefas desse nó em outros nós.

## Nodes

Um nó é uma instância do Docker que participa do Swarm.

Nó gerenciador:

* recebe definições de serviço
* dá tarefas para nós trabalhadores.
* executam as funções de orquestração e gerenciamento de cluster necessárias para manter o estado do swarm.

Nós do trabalhador:

* recebem e executam tarefas despachadas dos nós do gerenciador.

## Service

Tarefas a serem executadas nos nós do gerenciador ou do trabalhador. Um serviço é a raiz primária da interação do cliente com o enxame. Para serviços globais, o enxame executa uma tarefa para o serviço em cada nó disponível no cluster.

Um serviço:

* especifica qual imagem de contêiner usar
* especifica quais comandos executar
* no Swarm tem uma entrada DNS

## Load Balancing

O **swarm manager** usa load balancing para expor os serviços externamente ao swarm e interno para distribuir solicitações entre serviços dentro do cluster com base no nome DNS do serviço. O swarm manager atribui um PublishedPort ao serviço (ou nós podemos configurar um)

* load balancing em nuvem, podem acessar o serviço no PublishedPort de qualquer nó no cluster.
* os nós no swarm roteiam conexões de entrada para uma instância de tarefa em execução.

## Swarm services vs. standalone container

É possível modificar a configuração de um serviço, incluindo as redes e volumes aos quais está conectado, sem a necessidade de reiniciar manualmente o serviço. O Docker atualizará a configuração, interromperá as tarefas de serviço com a configuração desatualizada e criará novas que correspondam à configuração desejada.

# Kuebernetes

O Kubernetes é uma plataforma de código aberto para automatizar a implantação, dimensionamento e operações de contêineres de aplicativos.

* Horizontal Scalability
* Self-healing
* Service Discovery
* Automated Rollbacks

Init containers

Executados e concluídos antes que os contêineres do aplicativo sejam iniciados. Se um init contêiner falhar, o pod será reinicializado. Isso pode ser usado para fazer inicializações antes de iniciar o serviço principal.

# Objects

## Pod Resources

Encapsula:

* contêiner de uma app
* recursos de armazenamento
* IP de rede exclusivo
* opções que controlam como o(s) contêiner(es) deve(m) ser executado(s)

#### Networking

Cada pod recebe um endereço IP exclusivo. Cada contêiner em um pod compartilha o namespace da rede, incluindo o endereço IP e as portas de rede. Os contêineres dentro de um pod se comunicam usando localhost. Quando os contêineres em um pod se comunicam com entidades fora do pod, eles devem coordenar como usam os recursos de rede compartilhados (como portas).

#### Storage

Um pod pode especificar um conjunto de volumes de armazenamento compartilhado. Todos os contêineres no pod podem acessar os volumes compartilhados, permitindo que esses contêineres compartilhem dados. Os volumes também permitem que os dados persistentes em um pod sobrevivam caso um dos contêineres precise ser reiniciado.

## Services

Um serviço é uma abstração que define um conjunto lógico de pods e uma política para acessá-los. Por exemplo, considere um back-end de processamento de imagem sem estado que está sendo executado com 3 réplicas. Os frontends não se importam com qual backend eles usam e se eles mudam.

As APIs do Kubernetes para descoberta de serviço permitem consultar Endpoints, que são atualizados sempre que o conjunto de pods em um serviço muda. Para aplicativos não nativos, o Kubernetes oferece maneiras de colocar uma porta de rede ou um balanceador de carga entre seu aplicativo e os pods de back-end.

#### Tipos de serviços

**ClusterIP:** expõe o serviço em um IP interno do cluster (ServiceType padrão)

**NodePort:** Expõe o Serviço no IP de cada Node em uma porta estática (o NodePort). Um Serviço ClusterIP, ao qual as rotas do serviço NodePort são criadas automaticamente.

**LoadBalancer:** expõe o serviço externamente usando o balanceador de carga de um provedor de nuvem. Os serviços NodePort e ClusterIP, para os quais o balanceador de carga externo roteia, são criados automaticamente.

**ExternalName**: mapeia o serviço para o conteúdo do campo externalName

## Volumes

É um diretório acessível aos Containers em um Pod.

* quando um container falha, ele será reiniciado, e os arquivos serão perdidos
* ao executar contêineres juntos em um pod, é necessário compartilhar arquivos eles
* um volume do Kubernetes tem o mesmo tempo de vida do pod que o inclui.
* quando um Pod deixa de existir, o volume também deixa de existir.
* alguns volumes são persistentes

## Namespace

Namespaces fornecem um escopo para nomes. Os nomes dos recursos precisam ser exclusivos dentro de um namespace, mas não entre namespaces.

* o Kubernetes suporta vários clusters virtuais(namespaces) apoiados pelo mesmo cluster físico
* os namespaces destinam-se ao uso em ambientes com muitos usuários em vários projetos
* os nomes dos recursos precisam ser exclusivos dentro de um namespace, mas não entre namespaces.
* os namespaces não podem ser aninhados uns dentro dos outros e cada recurso do Kubernetes só pode estar em um namespace.

## Kubernetes controllers

#### Deployment

Um deployment fornece atualizações declarativas para pods e ReplicaSets. O deployment maneger altera o estado real para o estado desejado em uma taxa controlada.

#### ReplicaSet

Um ReplicaSet garante que um número especificado de réplicas de pod esteja em execução a qualquer momento. Um deployment gerencia ReplicaSets.

#### DeamonSet

Um DaemonSet garante que todos (ou alguns) Nodes executem uma cópia de um Pod. À medida que os nós são adicionados ao cluster, os pods são adicionados a eles. À medida que os nós são removidos do cluster, esses pods são coletados como lixo.

Nota: A exclusão de um DaemonSet limpará os pods que ele criou.

Alguns usos típicos de um DaemonSet são:

* executando um daemon de armazenamento de cluster, como glusterd, ceph, em cada nó.
* executar um daemon de coleta de logs em cada nó, como fluentd ou logstash.
* executar um daemon de monitoramento de nó em cada nó.

#### StatefulSet

StatefulSet é usado para gerenciar aplicativos com estado e é útil para aplicativos que exigem um ou mais dos seguintes itens:

* Identificadores de rede exclusivos e persistentes.
* Armazenamento persistente e persistente.
* Implantação e dimensionamento ordenados e otimizados.
* Atualizações contínuas automatizadas e ordenadas.

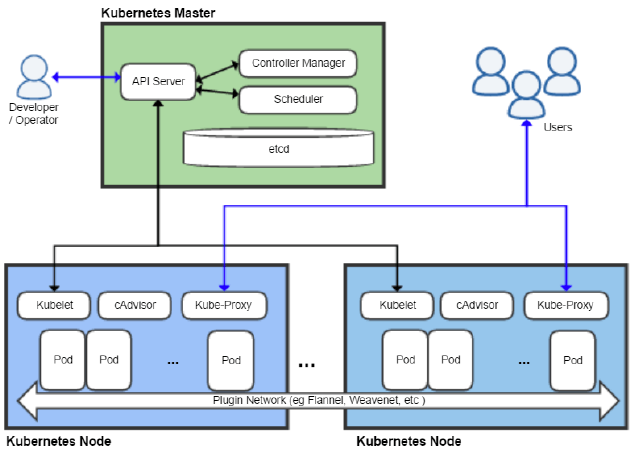
#### Job

Um trabalho cria um ou mais pods e garante que um número especificado deles seja encerrado com sucesso

## Kubernetes Control Plane

O Kubernetes Control Plane faz com que o estado atual do cluster corresponda ao estado desejado, executando uma variedade de tarefas automaticamente, como:

* iniciar ou reiniciar contêineres
* dimensionar o número de réplicas de um determinado aplicativo

O Kubernetes master é executado em um único nó de um cluster. E um nó do Kubernetes é:

* kubelet 🡪 que se comunica com o Kubernetes Master
* kube-proxy 🡪 um proxy de rede que reflete os serviços de rede Kubernetes em cada nó.

# Kubernetes vs. Swarm

Swarm funciona 🡪 apenas com Docker

Kubernetes funciona 🡪 com Docker e outros serviços de contêiner

O Kubernetes é mais complexo de implementar e gerenciar em comparação com o Swarm.

Kubernetes aparenta ter melhor escalabilidade.