**1) A utilização de máquinas virtuais e/ou *containers* é bastante comum por parte dos fornecedores de serviços de computação em nuvem. De que forma estas tecnologias permitem reduzir custos de infraestrutura e aumentar a elasticidade destes serviços?**

A virtualização é o único motivo pelo qual é possível aos fornecedores de soluções de cloud-computing gerir dezenas de milhares de servidores:

-Recursos virtuais não têm problemas de compatibilidade com hardware podendo correr sem problemas em máquinas diferentes.

-A interação de um utilizador com um recurso virtual é semelhante à com um físico.

-Os recursos virtuais estão isolados das suas infraestruturas físicas não comprometendo segurança, performance e falhas.

-A utilização de recursos virtuais permite uma utilização otimizada dos recursos físicos conforme as necessidades de processamento/memória.

-Os recursos virtuais têm uma gestão muito simplificada e possuem ferramentas muito úteis para caso haver necessidade de migração.

**2) Na medição da desempenho de sistemas de gestão e armazenamento de dados o tempo de res- posta é uma das principais métricas utilizadas. Para esta métrica devem ser apresentadas médias ou percentis? Justifique.**

Percentil, porque a média oculta quantos utilizadores sofreram de um eventual atraso, enquanto que com percentil podemos observar diferentes valores de percentil.

**3) Quais as vantagens/desvantagens do uso de replicação síncrona vs assíncrona?**

Quando temos síncrona:

-Assegura consistência.

-Tempos de espera para leitura possivelmente mais elevados.

-Assíncrona pode causar registos desatualizados.

**4) Quais as principais vantagens de *Software Defined Storage*?**

A leitura e a escrita estão completamente separadas da componente de controlo, o que permite ter um controlo lógico e centralizado.

A componente dos dados é completamente programável e extensível.

A componente de controlo é distribuído e adaptativa, para além de que permite assegurar umas configurações de leitura e escrita muita úteis, como transformação, gestão e reencaminhamento e ferramentas como priorização, limitação de acessos, encriptação etc.

**5) Descreva de forma sucinta as principais fases usadas pelos motores de bases de dados na optimização de interrogações.**

1. Calcular o custo de uma expressão de álgebra relacional.
2. Conversão de uma expressão algébrica para outra
3. Escolher paths de acesso para avaliar subexpressões (o que quer que isto signifique…)

**6) Compare de forma sucinta os sistemas de gestão de bases de dados com os sistemas de gestão de *streams*.**

Ora bem esta pergunta é um bocado estúpido porque tão nos a pedir para comparar alhos com bugalhos, mas alguém achou boa ideia fazer um slide sobre isto, portanto vamos tentar fazer isto de um forma memorável.

Sistemas de gestão de streams:

-Os relacionamentos alteram-se ao longo do tempo

-Os relacionamentos são representados através de sequências de tuplos

-Os updates são através de appends

-As respostas das queries são apenas aproximações

Sistemas de gestão de bd's;

-Relacionamentos persistentes

-Relacionamentos representados através de conjuntos/bags de tuplos

-Os updates são modificações mesmo.

-As respostas das queries são precisas

**RESUMOS**

**Vantagens da virtualização:**

* transparência
* isolamento
* optimização de recursos
* gestão simplificada

**Desvantagens da virtualização:**

* performance
* overprovisioning
* segurança
* dependability
* learning curve

IaaS - Infrastructure as a Service (hardware + resources) ex: VMs

PaaS - Platform as a Service (encapsulation of dev. env.) deployment items ex: RedshiftSQL

SaaS - Software as a Service (full apps) no deployment ex: Salesforce, Gmail

**Vantages IaaS:**

* conveniência
* velocidade
* elasticidade
* preço flexível

**Desvantages IaaS:**

* perda de controlo
* segurança
* privacidade

**Ao passar de IaaS para PaaS:**

Deixa de haver uma gestão de recursos por parte do utilizador mas sim por parte do sistema em si, os recursos passam a ser transparentes ao utilizador, providenciando as conexões a outros IaaS e SaaS necessários para que o utilizador apenas se tenha de focar no desenvolvimento do produto em causa.

**Containers:**

Virtual environments leves que agrupam um conjunto de processos e recursos do anfitrião. Hardware & kernel/os é partilhado com o sistema anfitrião.

**Utilidade dos containers:**

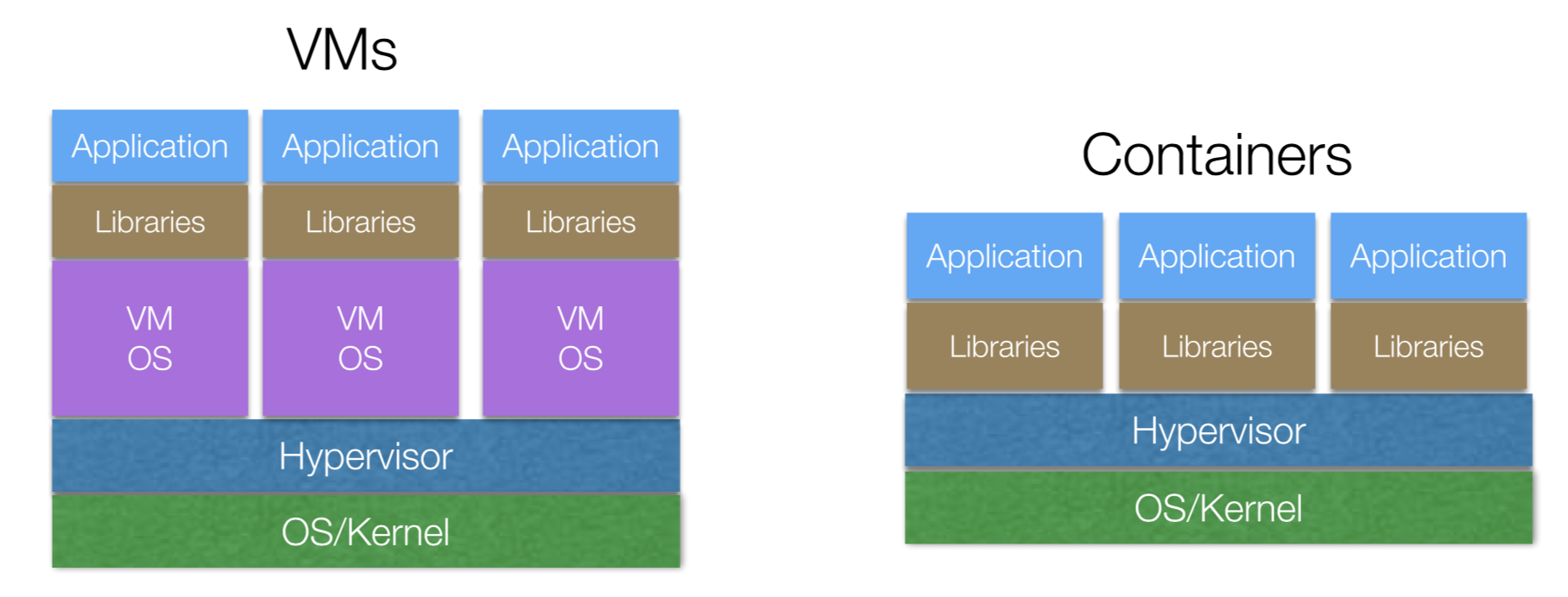
* Permitem correr diferentes versões do mesmo software no mesmo host, em simultâneo. Tipo DBs.
* Migração é facil para xuxu
* Também dá para empacotar logo tudo com as dependências e pumba

**Vantagens dos containers vs VMs:**

* mais rápidos e mais fáceis usar
* melhor optimização de recursos

**Desvantagens dos containers vs VMs:**

* isolamento mais fraco
* não tem flexibilidade em relação ao SO



**Tipos de falhas:**

**Hardware:**

No processador: paragens, atrasos, resets. Na memória podem ocorrer escritas atómicas não supostas e falhas de rede

**Software:**

Podem ocorrer erros sistemáticas, erros cujo dada um determinado input poderão resultar numa falha nos serviços em escala. Utilização de memória partilhada. Falhas em cascata, em uma falha num componente despoleta uma falha noutro e por aí adiante.

**Falhas humanas:**

Representam a gigantesca maioria das falhas

**Porquê a necessidade de sistemas distribuídos:**

* Escalabilidade - se o teu volume de dados/processamento exceder aquilo que apenas uma máquina consegue processar
* Tolerância a falhas - se não quiseres que estoure tudo porque um componente ou um servidor estourou
* Latência - Se os teus utilizadores estão distribuídos é sempre melhor as tuas máquinas também estarem distribuídas e pertinho deles, não é?

**Replication VS Partitioning:**

**Replication:** replicação dos dados de uma forma redundante ao longo de diversos nodos, de forma a trazer melhorias de performance e tolerancia a falhas. Traz melhoria de perfomance porque pode manter os dados geograficamente proximos dos utilizadores. Tolerancia a falhas porque previne todos os dados necessários estarem em apenas nodo que falhou. E também escalabilidade porque permite um maior throughput no de queries de leitura. Como é obvio tem o problema da consistência, mas se os dados forem imutáveis tá resolvido!.

**Partitioning:** repartição dos dados ao longo dos nodos

**Bem, mas como funciona a replicação:**

**Statement-based replication**: o leader faz os seus writes e depois envia log dos mesmos para os outros nodos e eles que se desamerdem

**Write-ahead log (WAL) shipping**: existe apenas um log! que é binário com todos os writes e é escrito apenas com append. É fixe porque dá para criar uma replicação total do sistema assim de estoura, basta o novo nodo ter o log.

**Logical (row-based) log replication:** é replicado ao log? parece que sim.. É fixolas para a aplicações externas fazerem parse porque entende-se bem né.

**Trigger-based replication:** fazer as alterações através de triggers, mas isto está muito sujeito a erros.

**Replicação passiva/leader-based:**

Somente replicação primária recebe as alterações, atualiza-se e envia mensagem às outras. As outras atualizam-se, informam a primária e a primária avisa o utilizador que tá tudo ok. Os writes vao todos para um nodo mas os reads podem ir para qualquer um que é na boa. - Pode originar “replication lag”, ou seja, pode haver reads de valores, nas réplicas, que ainda não tenham sido atualizados. Exige Ordem Causal das mensagens (Relógios Lógicos) no modelo Assíncrono. Em síncrono não importa, porque é tudo seguidinho.

**Active replication:**

Na ativa, o cliente manda para todos um pedido, mas tem de esperar pelas respostas de todas as máquinas para efetivamente ter sucesso. Como o pedido é “independente” para cada uma das máquinas, tem de ser deterministico (tem de dar o mesmo resultado em todas). Por outro lado, a vantagem para a passiva, é que se uma máquina falhar, tende a ser transparente para o cliente - de alguma forma ele sabe que falhou e deixa de esperar pela resposta dela, aceitando as outras. Na passiva, no caso de ser o líder a estourar existe uma complicação muito maior, desde a necessidade de eleger um novo lider, retransmitir pedidos se necessário, ou recuperar estados de forma a saber a “consistência” das restantes máquinas que não eram o lider. Exige Ordem Total das mensagens.

**Relógios:**

Os relógios nos sistemas distribuídos fodem tudo…. Cada máquina tem o seu, existem delays nas mensagens e ainda os fuso-horários. Como é que se resolve? Relógios lógicos, oh yeah! Funcionam à base de contadores incrementáveis e apenas representam a ordem dos eventos.

- representam o conhecimento (valor incrementável) que têm sobre outras máquinas. dessa forma consegue-se perceber se existem mensagens antecedentes ou com uma relação causal entre elas. Por exemplo:

A: Tudo Bem?

B: Sim e contigo?

mas C, ler => “Sim e contigo?” “Tudo Bem?”. Com relógios lógicos, isto não acontece, independentemente de a mensagem de B ser entregue primeiro que a de A..

**HADOOP:**

**NameNode:** É o que comunica com o cliente, mantém os metadados do filesystem. Gere todos as atividades do sistema, como os locks essas cenas fixes

**DataNode:** Guarda dados e devolve-os quando é preciso… Tudo repartidinho em blocos se é que isso interessa. Blocos de 64MiB ou 128MiB. E podem estar replicados entre nodos como já sabemos.

With HBase there's a region server for each datanode and a namenode for the master.

**Eventual consistency:**

É o princípio que todas as BD's replicadas eventualmente estarão consistentes. Basta esperar um bocado que cada uma se atualizará e os reads já darão todos iguais (convergir). O problema é que não se sabe quando é que vai haver essa convergência.

**Online Processing:**

Um serviço aguarda um pedido de um cliente e assim que o recebe tenta responder o mais rápido possível. A métrica de performance mais relevante aqui é o tempo de resposta, mas também availability.

**Batch processing:**

Recebe um (possívelmente enorme) conjunto de dados, processa-o e escreve outro conjunto de dados, os resultados. Por vezes duram imenso, normalmente não há um utilizador há espera dos resultados. Normalmente são programados, tipo para acontecer diáriamente (por exemplo). A métrica de interesse aqui é o throughput (quantas unidades são processados por minuto).

**Stream processing:**

A diferença entre para batch processing é que batch processing recebe de uma vez só um determinado número de dados e mastiga tudo, stream processing vai recebendo evento a evento e processando, o que permite ter uma menor latência pois tem os resultados mais a pronto.

**Como funcionam Dataflow engines como o Spark?:**

Em vez de tratar o processo todo como "pequenos" jobs independentes, trata-o como um todo. Ele paraleliza os processos, particiona os inputs e envia os outputs das funções para input de outras. Os outputs do mapreduce são escritos para ficheiros temporários enquanto que os dados no spark deslocam-se mais ao estilo de pipelines UNIX.

Vantagens: não tem que adotar o estilo restritivo de fazer map e reduce nem tendo que obrigatoriamente fazer o sorting de uma etapa para a outra, o que é uma operação muito custosa.

Como o processo tem noção das dependências de dados o scheduler consegue fazer otimização.

É suficiente o estado intermédio ser guardado em ficheiro em vez de no HDFS, o que é muito mais rápido.

Funções podem começar a executar a sim que o seu input estiver pronto, não há necessidade de ficar à espera que todo o processo anterior termine.

Podes usar os mesmo JVM sem ter que estar sempre a inicializar os programas, o que poupa muito no overhead.

A cena do mapreduce escrever estados intermédios no HDFS torna a tolerância à falha bastante fácil. Basta ler e continuar. O Spark tem que lidar com a tolerância à falha através de outro método: os dados perdidos são recalculados a partir de dados de estados anterior que ainda existam, mas para isto a framework tem de andar a controlar onde é que eles andam a toda a hora.

O Flink é fixe porque como passa os dados de um lado para o outro tipo pipeline, um operador nem precisa de receber o input todo que já da para começar a processar. A não ser, claro, que seja algo tipo sort, que requer o input todo antes de começar a dar output.

**PUB/SUB**

**Se os producers enviam mensagens com mais frequencia do que os subscribers as lêem, temos várias hipóteses:**

* Fazer drop
* Por numa queue
* bloquear o producer, mm à fdp

**Quando há multiplos consumidores/subscribers há 2 tipos de broker:**

1. Load balancing, que basicamente cada mensagem é lida por apenas um utilizador
2. Fan-out, que cada mensagem é enviada para todos os utilizadores.

**Como extrair informação/relações útil e finita dos streams se os streams nunca acabam?**

Criando janelas, definindo o conjunto de dados de forma finita. Podem ser janelas temporais, em função de atributos ou até de outros símbolos, por exemplo pontuação numa string. Windows:

* Tumbling window: tamanho fixo, com cada evento a pertencer a uma e só uma window ("dá me intervalos de 10 em 10 segundos")
* Hopping window: tamanho fixo, mas um evento pode pertencer a mais que uma. ("a cada 5 segundos dá me os ultimos 10 segundo")
* Sliding window: e.g.: ("Dá me as listas dos tópicos que foram escritos mais de 10x nos ultimos 15s")
* Session window: sem tamanho fixo e.g.: se um utilizador ficar inativo por X tempo a sessao acaba e a window também ("dá os conjuntos de tweets que aconteceram num intervalo de 15 segundos uns dos outros")

**Privacy design strategies:**

Orientadas aos dados:

Minimise - limita ao maximo a quantidade de dados pessoais processados

Separate - Não ponhas dados pessoais juntos de outros dados úteis. Põe nos lá bem longe

Abstract - limite o detalhe com que os dados pessoais são processados

Hide - Põe tudo de forma a que se torne o menos possível ficarem públicos. Criptografia, etc.

Orientadas aos processos:

Informar - informar o utilizador de como vão ser utilizados os dados

Controlar - dar ao utilizador possibilidade de os editar