**Pontos Importantes Prova Banco de Dados NoSQL – UFU**

**Por que NoSQL?**

**A Jornada dos Dados: Da Persistência Clássica à Arquitetura Moderna**

No universo do desenvolvimento de software, a persistência é o alicerce que garante a longevidade dos dados. Ela representa a capacidade de uma aplicação de salvar suas informações de um estado volátil (como a memória RAM, que se apaga ao final do processo) para um meio de armazenamento duradouro e não volátil, como um disco rígido ou SSD. Em essência, é o que permite que os dados "sobrevivam" ao ciclo de vida da aplicação que os criou, estando disponíveis para uso futuro.

**O Desafio Clássico: A Incompatibilidade de Impedância Objeto-Relacional**

Historicamente, o principal desafio na implementação da persistência surgiu de um conflito de paradigmas. A grande maioria das aplicações é desenvolvida utilizando a **Orientação a Objetos (OO)**, que modela o mundo real através de objetos, com seus atributos, métodos e relacionamentos complexos (herança, polimorfismo, encapsulamento). Por outro lado, o pilar do armazenamento de dados por décadas tem sido os **Bancos de Dados Relacionais (RDBMS)**, que organizam as informações em um modelo tabular rígido, composto por tabelas, linhas e colunas.

Essa "diferença de mundos" gera o que é conhecido como **incompatibilidade de impedância objeto-relacional**. Trata-se de um atrito fundamental que se manifesta em diversos problemas:

* **Complexidade no Mapeamento:** Como traduzir uma hierarquia de classes em um conjunto de tabelas?
* **Aumento do Custo de Desenvolvimento:** Programadores precisam gastar tempo escrevendo código repetitivo (conhecido como *boilerplate*) para converter dados entre os dois modelos.
* **Problemas de Desempenho:** Consultas complexas para reconstruir objetos a partir de múltiplas tabelas (operações de JOIN) podem se tornar gargalos de performance.

Para mitigar esse atrito, surgiram as ferramentas de **Mapeamento Objeto-Relacional (ORM)**, como Hibernate (Java), Entity Framework (.NET) e SQLAlchemy (Python). Elas atuam como uma "ponte", automatizando a tradução entre os objetos da aplicação e as tabelas do banco de dados, permitindo que o desenvolvedor manipule os dados utilizando os conceitos da orientação a objetos, enquanto o ORM cuida da complexidade da interação com o banco de dados relacional.

**A Evolução: Persistência Poliglota e o Universo NoSQL**

O cenário de dados, no entanto, tornou-se muito mais diverso. Nem todo dado se encaixa perfeitamente no modelo relacional. Aplicações modernas lidam com grandes volumes de dados não estruturados ou semiestruturados, como documentos JSON, grafos de redes sociais ou dados de séries temporais.

Nesse contexto, emerge a **persistência poliglota**, uma abordagem de arquitetura de software que defende uma ideia poderosa: **usar a ferramenta certa para o trabalho certo**. Em vez de forçar um único tipo de banco de dados a resolver todos os problemas, uma aplicação pode utilizar múltiplas tecnologias de armazenamento, cada uma otimizada para uma necessidade específica.

Essa abordagem foi amplamente impulsionada pela ascensão dos bancos de dados **NoSQL** (acrônimo para "*Not Only SQL*"). Diferente dos RDBMS, os bancos NoSQL geralmente operam sem um esquema fixo (*schema-less*), oferecendo flexibilidade para adicionar ou alterar campos nos registros sem a necessidade de modificar a estrutura de uma tabela previamente definida. Eles se dividem em várias categorias, cada uma com seus pontos fortes:

* **Bancos de Documentos (ex: MongoDB, CouchDB):** Armazenam dados em formatos como JSON ou BSON, ideais para catálogos de produtos, perfis de usuário e conteúdos diversos.
* **Bancos Chave-Valor (ex: Redis, Riak):** Modelos de altíssima performance para armazenar dados simples, muito usados para caches de sessão e dados temporários.
* **Bancos de Grafos (ex: Neo4j, Amazon Neptune):** Especializados em armazenar e navegar por relacionamentos complexos, perfeitos para redes sociais, sistemas de recomendação e detecção de fraudes.
* **Bancos Orientados a Colunas (ex: Cassandra, HBase):** Otimizados para consultas analíticas em grandes volumes de dados (Big Data).

**A Mudança de Paradigma na Integração: De Dados a Serviços**

Finalmente, há uma mudança fundamental na forma como os sistemas se integram. O antigo modelo de usar um **banco de dados compartilhado como ponto de integração** entre múltiplas aplicações mostrou-se problemático, criando um acoplamento forte e dificultando a evolução independente dos sistemas. Se uma aplicação precisa de uma mudança no esquema do banco, todas as outras que o acessam são potencialmente impactadas.

A tendência moderna, alinhada com arquiteturas como os **microsserviços**, é o **encapsulamento total do banco de dados dentro do serviço ou aplicação que é seu "dono"**. A integração não ocorre mais no nível dos dados, mas sim por meio de **APIs e serviços bem definidos**. Cada serviço expõe suas funcionalidades através de uma interface clara (como uma API REST ou gRPC), e é o único responsável por seu próprio banco de dados. Essa abordagem garante:

* **Autonomia dos Times:** Equipes podem evoluir, escalar e substituir a tecnologia de persistência de seus serviços sem impactar o resto do sistema.
* **Flexibilidade Tecnológica:** Favorece a persistência poliglota, pois cada microsserviço pode escolher o banco de dados que melhor se adapta às suas necessidades.
* **Maior Resiliência e Escalabilidade:** Isola falhas e permite que cada componente do sistema seja escalado de forma independente.

Em suma, a jornada da persistência de dados reflete a própria evolução da engenharia de software: uma busca contínua por maior flexibilidade, desacoplamento e o uso de ferramentas especializadas para construir sistemas mais robustos, escaláveis e fáceis de manter.

**Modelos de Dados Agregados**

**O que é um Agregado (Aggregate)?**

O termo **Agregado (Aggregate)** tem sua origem no **Domain-Driven Design (DDD)**, uma abordagem de design de software criada por Eric Evans.

Em sua essência, um Agregado é um cluster ou uma coleção de objetos de domínio relacionados que devem ser tratados como uma única unidade. O objetivo principal é garantir a consistência e a integridade das regras de negócio.

Um agregado possui duas características centrais:

* **Raiz do Agregado (Aggregate Root):** É uma entidade específica dentro do agregado que serve como o único ponto de entrada para qualquer modificação nos dados contidos nele. Qualquer referência externa só pode apontar para a raiz.
* **Fronteira de Consistência (Consistency Boundary):** A fronteira do agregado define o que está "dentro" e o que está "fora". Todas as operações que alteram o estado do agregado devem ser atômicas. Ou seja, ou a operação é concluída com sucesso para todo o agregado, ou ela falha por completo, não deixando os dados em um estado inconsistente.

**Exemplo Prático: Um Pedido Online**

Imagine um sistema de e-commerce. Um Pedido pode ser um excelente exemplo de agregado.

* **Raiz do Agregado:** A entidade Pedido.
* **Objetos Internos:** ItemPedido (uma lista de produtos), EnderecoEntrega, StatusPedido.
* **Fronteira de Consistência:** O objeto Pedido junto com seus itens e endereço forma uma unidade. Você não pode adicionar um ItemPedido diretamente; você deve fazê-lo através do objeto Pedido (ex: pedido.adicionarItem(...)). Isso permite que a raiz do agregado valide regras de negócio, como "não é possível adicionar itens a um pedido já enviado".

**Agregados e a Ascensão do NoSQL**

A principal razão para a popularidade da orientação a agregados em bancos de dados NoSQL está diretamente ligada à necessidade de **escalabilidade em clusters distribuídos**.

Em um cluster, os dados são particionados e distribuídos por vários servidores (nós). Se os dados relacionados a uma única transação estiverem espalhados por múltiplos nós, o banco de dados precisará coordenar a operação entre eles. Essa coordenação é complexa, lenta e um grande gargalo para a performance.

O agregado resolve isso de forma elegante. Ao modelar os dados como agregados, damos uma dica explícita ao banco de dados: **"todos estes dados devem ser manipulados juntos"**. Consequentemente, o banco de dados garante que todo o agregado (como o objeto Pedido do exemplo acima) seja armazenado no mesmo nó.

Quando você precisa ler ou atualizar um pedido, a operação é direcionada a um único nó, tornando-a extremamente rápida e eficiente, pois minimiza a comunicação pela rede.

**Transações, Consistência e o "Agregado-Ignorante"**

É uma simplificação excessiva e, em muitos casos, incorreta, afirmar que "bancos de dados NoSQL não suportam transações ACID". A verdade é mais sutil:

* **Bancos de dados orientados a agregados** (a maioria dos NoSQL) **suportam transações ACID no escopo de um único agregado**. Você pode atualizar um pedido e seus itens de forma atômica e consistente.
* O que eles geralmente **não suportam** são transações ACID que abrangem **vários agregados**. Por exemplo, uma única transação atômica que atualiza um Pedido e deduz o estoque de múltiplos Produtos (que seriam outros agregados) geralmente não é possível nativamente. Nesses casos, a gestão da consistência precisa ser tratada na aplicação, muitas vezes usando padrões como Sagas.

Aqui entra o conceito de **agregado-ignorante (aggregate-ignorant)**:

* **Agregado-Ignorante:** Um sistema que não reconhece o conceito de agregado como uma fronteira de consistência. Bancos de dados relacionais são classicamente agregado-ignorantes. Eles permitem que você crie uma transação que una e modifique quaisquer tabelas (JOIN, UPDATE), independentemente das fronteiras lógicas do domínio.
* **Orientado a Agregado:** Um sistema que entende o agregado como a unidade fundamental de armazenamento e consistência. A maioria dos bancos NoSQL se encaixa aqui.

**Como Diferentes Tipos de NoSQL Lidam com Agregados**

O conceito de agregado é a espinha dorsal comum, mas a forma de interagir com ele varia entre os tipos de bancos NoSQL.

* **Bancos de Dados Chave-Valor (Key-Value):** Este é o modelo mais simples. O agregado é o "valor", geralmente armazenado como uma string, JSON ou outro formato serializado. Ele é indexado por uma "chave" única (como "pedido\_12345"). A principal forma de acesso é obter o agregado inteiro usando sua chave.
* **Bancos de Dados de Documentos (Document Databases):** São uma evolução natural dos bancos chave-valor. O agregado é armazenado como um documento (tipicamente JSON ou BSON), mas, crucialmente, o banco de dados entende a estrutura interna do documento. Isso permite consultas muito mais ricas e flexíveis. Em vez de buscar apenas pela chave principal, você pode buscar por qualquer campo dentro do agregado, como: *"encontre todos os pedidos com status: 'Processando'"*.
* **Bancos de Dados de Família de Colunas (Column-Family Databases):** Sistemas como Bigtable, Cassandra e HBase utilizam o conceito de agregado, mas o organizam em "famílias de colunas". Isso pode ser visto de duas maneiras:
  + **Visão Lógica (Orientada a Linha):** Para o desenvolvedor, um agregado (ex: um Cliente com ID 1234) é uma "linha". Dentro dessa linha, os dados são agrupados em famílias de colunas, como perfil (com nome, email) e historico\_pedidos (com uma lista de IDs de pedidos).
  + **Visão de Armazenamento (Orientada a Coluna):** Fisicamente, o banco de dados armazena os dados por família de colunas. Isso significa que os dados da família perfil de todos os clientes pode ser armazenados juntos, otimizando consultas que precisam apenas dessa informação específica (ex: "obter o e-mail de 1000 clientes").

**Em resumo:**

* Use **Chave-Valor** se você sempre vai buscar o agregado inteiro pelo seu ID.
* Use **Documento** se você precisa de flexibilidade para buscar agregados com base em diferentes informações contidas neles.
* Use **Família de Colunas** se seus agregados são enormes e você geralmente só precisa acessar partes específicas deles de cada vez.

Independentemente do modelo, o princípio unificador permanece: **o agregado, indexado por uma chave, é a unidade central de distribuição e consistência**, garantindo que os dados que mudam juntos permaneçam juntos, o que é essencial para a escalabilidade no mundo distribuído.

**Mais detalhes sobre modelos de dados**

**O Desafio das Relações: Lidando com Múltiplos Agregados**

Quando modelamos um sistema, raramente um agregado vive isolado. Um Pedido precisa se relacionar com um Cliente e com Produtos. Entender como gerenciar essas relações é crucial e revela as maiores diferenças entre os bancos de dados.

**Relacionamentos e o Dilema das Atualizações**

Em um banco de dados orientado a agregados, a relação entre eles é geralmente feita por referência. Por exemplo, um agregado Pedido não contém o objeto Cliente inteiro dentro de si; ele armazena apenas uma referência, como o clienteId.

JSON

// Agregado Pedido

{

"pedidoId": "PED123",

"clienteId": "CLI456", // Referência para outro agregado

"status": "Processando",

"itens": [

{ "produtoId": "PROD789", "quantidade": 2 } // Referência

]

}

O grande desafio surge ao atualizar múltiplos agregados ao mesmo tempo.

**O princípio fundamental é: a atomicidade (a garantia do "tudo ou nada") só é suportada dentro de um único agregado.**

**Exemplo: O Pedido e o Estoque**

Imagine que um cliente faz um pedido. Duas coisas precisam acontecer:

1. Um novo agregado Pedido deve ser criado.
2. O agregado Produto correspondente deve ter seu estoque decrementado.

Em um banco de dados NoSQL orientado a agregados, não há uma transação ACID única que possa garantir que ambas as operações aconteçam. Se o sistema criar o pedido mas falhar antes de atualizar o estoque, você terá uma inconsistência grave.

* **Solução NoSQL (Consistência Eventual):** Isso é geralmente resolvido na camada de aplicação, usando padrões como **Sagas**. A aplicação primeiro confirma o pedido e depois dispara um evento para um outro serviço atualizar o estoque. O sistema fica "eventualmente consistente". Há um risco e uma complexidade a serem gerenciados.
* **Solução Relacional (ACID):** Em um banco de dados relacional, isso é simples. Você inicia uma transação, insere a linha na tabela pedidos, atualiza a linha na tabela produtos e confirma a transação. O banco garante que ou ambas as operações funcionam, ou nenhuma delas.

**Visualização e Consultas Através de Agregados**

Se os agregados são otimizados para serem lidos individualmente, como realizamos consultas que precisam de dados de múltiplos agregados? Por exemplo: "Mostrar os 10 últimos pedidos com o nome do cliente e o nome dos produtos".

Fazer JOINs em tempo real em um cluster distribuído é uma operação cara e lenta, por isso a maioria dos bancos orientados a agregados a evita. A solução mais comum é criar **Visões Materializadas (Materialized Views)**.

Uma Visão Materializada não é uma consulta dinâmica; é, na prática, uma **tabela pré-calculada e otimizada para leitura**.

**Exemplo: Criando uma Visão de Pedidos Detalhados**

1. **Agregados Base:** Temos os agregados Cliente, Pedido e Produto.
2. **Criação da Visão:** Criamos uma nova coleção/tabela chamada VisaoPedidosDetalhados.
3. **Processo:** Sempre que um novo Pedido é criado ou atualizado, um processo em background (ou um gatilho) é executado. Ele lê as informações necessárias do Pedido, busca o nome do Cliente pelo clienteId e o nome do Produto pelo produtoId.
4. **Armazenamento:** Ele então armazena um novo documento "desnormalizado" na VisaoPedidosDetalhados.

JSON

// Documento na Visão Materializada "VisaoPedidosDetalhados"

{

"pedidoId": "PED123",

"data": "2025-07-15",

"nomeCliente": "Maria Silva", // Dado vindo do agregado Cliente

"nomeProduto": "Notebook Gamer X", // Dado vindo do agregado Produto

"quantidade": 2,

"status": "Enviado"

}

Agora, quando a aplicação precisa mostrar o histórico de pedidos, ela faz uma consulta simples e extremamente rápida diretamente a essa visão, sem a necessidade de JOINs complexos.

**Uma Abordagem Diferente: Bancos de Dados de Grafos**

Bancos de dados de grafos são "um peixe estranho no lago NoSQL". Eles não surgiram da necessidade de gerenciar grandes agregados em clusters, mas sim da frustração com a dificuldade dos bancos relacionais em lidar com relacionamentos complexos.

O modelo de dados de um grafo é radicalmente diferente:

* **Nós (Nodes):** Representam as entidades (um Cliente, um Produto).
* **Arestas (Edges):** Representam os **relacionamentos** entre os nós. Crucialmente, os relacionamentos são entidades de primeira classe, com nome e propriedades.

**Filosofia Oposta:**

* **Orientado a Agregado:** Grandes registros com conexões simples (referências).
* **Orientado a Grafo:** Pequenos registros com interconexões ricas e complexas.

**Exemplo: Encontrando Recomendações**

Imagine a consulta: "Encontre todos os clientes que moram na mesma cidade que o 'João' e que compraram o mesmo produto que ele".

* **No modelo de Agregado/Relacional:** Seria uma consulta complexa com múltiplos JOINs.
* **No modelo de Grafo:** Seria uma "travessia" natural pelo grafo:
  1. Encontre o nó Cliente 'João'.
  2. Siga a aresta MORA\_EM para encontrar o nó Cidade.
  3. A partir do nó Cidade, encontre todos os outros Clientes com uma aresta MORA\_EM de entrada.
  4. Para cada um desses clientes, navegue pela aresta COMPROU para ver se eles se conectam ao mesmo nó Produto que o João.

Devido a essa ênfase nas relações, os bancos de grafos frequentemente priorizam a consistência das transações através de múltiplos nós e arestas, o que muitas vezes significa que são executados em um único servidor ou usam estratégias de cluster diferentes.

**A Flexibilidade do "Sem Esquema" (Schemaless)**

Uma característica comum a muitos bancos NoSQL (sejam de documentos, grafos, etc.) é a natureza "sem esquema". Isso não significa ausência de estrutura, mas sim que o banco de dados não impõe um esquema rígido.

* **O que isso permite?** Você pode facilmente adicionar novos campos a um registro (um documento, um nó) sem precisar alterar uma estrutura central. Por exemplo, você pode adicionar um campo "preferenciasDeMarketing" a alguns documentos de Cliente sem afetar os outros. Se depois decidir não usar mais, basta parar de gravá-lo.
* **O Alerta:** Embora o banco não exija um esquema, **sua aplicação quase sempre espera um**. A flexibilidade é uma grande vantagem durante o desenvolvimento e a evolução de um projeto, mas é fundamental manter um "esquema implícito" para que os dados permaneçam consistentes e utilizáveis.

**Modelos de Distribuição**

**Escalabilidade e Resiliência em Bancos de Dados NoSQL**

O principal motivador para a adoção de bancos de dados NoSQL tem sido sua capacidade de escalar horizontalmente, ou seja, de executar em um grande cluster de servidores. Vamos entender como isso funciona.

**O Desafio: A Necessidade de Escalar**

Quando o volume de dados e acessos cresce, um único servidor se torna insuficiente. Existem duas formas de escalar:

* **Escalabilidade Vertical (Scale-Up):** Comprar um servidor maior, mais potente e mais caro. Isso tem um limite físico e financeiro.
* **Escalabilidade Horizontal (Scale-Out):** Distribuir a carga de trabalho em um cluster de servidores mais baratos e comuns. Esta é a abordagem preferida para sistemas modernos e de grande porte.

Para escalar horizontalmente, os dados precisam ser distribuídos pelo cluster. Existem duas estratégias fundamentais para fazer isso: **Sharding** e **Replicação**.

**O que é Resiliência?**

Antes de detalhar as estratégias, é crucial entender um dos seus maiores benefícios: a **resiliência**.

**Resiliência é a capacidade de um sistema continuar funcionando mesmo quando um ou mais de seus componentes falham.**

No contexto de um banco de dados em cluster, isso significa que se um servidor (nó) parar de funcionar (por uma falha de hardware, por exemplo), o sistema como um todo não para. As outras cópias dos dados em outros nós assumem a responsabilidade, e as aplicações dos usuários continuam a ser atendidas, muitas vezes sem sequer perceberem que houve uma falha.

**As Estratégias de Distribuição**

Sharding e Replicação são técnicas distintas que podem ser usadas separadamente ou, mais comumente, em conjunto.

**Sharding (Fragmentação)**

**Sharding consiste em particionar os dados em subconjuntos menores (chamados de shards) e armazenar cada shard em um servidor diferente.**

Pense em uma grande enciclopédia. Em vez de um único volume gigante (difícil de carregar), você a divide em volumes de A-D, E-H, etc. Cada volume é um shard.

* **O Papel do Agregado:** É aqui que o conceito de **agregado** se torna fundamental. Como o agregado já agrupa dados que são acessados juntos (um Cliente com todos os seus dados, um Pedido com todos os seus itens), ele se torna a unidade natural para o sharding. O banco de dados garante que um agregado inteiro resida em um único shard.
* **Exemplo de Sharding:** Um banco de dados de clientes pode ser dividido usando a regiaoGeografica como chave de sharding. Todos os clientes do "Brasil" vão para o Shard 1 (no data center da América do Sul), enquanto todos os clientes da "Europa" vão para o Shard 2 (no data center de Frankfurt).
* **Principal Vantagem:** O sharding é excelente para **escalar operações de escrita (writes)**. Como as escritas para clientes de diferentes regiões vão para servidores diferentes, elas podem acontecer em paralelo, aumentando massivamente a capacidade do sistema.

**Replicação (Cópia de Dados)**

**Replicação consiste em copiar os mesmos dados e mantê-los em múltiplos servidores (nós).**

Seu principal objetivo é garantir a **resiliência** e a **escalabilidade de leitura (reads)**. Existem duas abordagens principais:

**A) Replicação Mestre-Escravo (Master-Slave)**

Neste modelo, os nós têm papéis diferentes:

* **Mestre (Master/Primary):** Há um único nó mestre. Ele é a fonte oficial da verdade e **recebe todas as operações de escrita**.
* **Escravos (Slaves/Secondaries):** Os outros nós são escravos. Eles recebem uma cópia dos dados do mestre e são usados para **servir operações de leitura**.

**Vantagens:**

* **Resiliência:** Se o mestre falhar, um dos escravos pode ser promovido a novo mestre rapidamente, garantindo a continuidade do serviço (failover).
* **Escalabilidade de Leitura:** As leituras podem ser distribuídas entre vários escravos, aliviando a carga do mestre.

**Desvantagens:**

* **Gargalo de Escrita:** Todas as escritas ainda precisam passar por um único mestre, o que limita a escalabilidade de escrita.
* **Inconsistência Temporária (Replication Lag):** Pode haver um pequeno atraso (lag) até que uma alteração feita no mestre seja replicada para todos os escravos. Isso significa que, por um breve momento, uma leitura em um escravo pode retornar um dado desatualizado.

**B) Replicação Ponto a Ponto (Peer-to-Peer)**

Neste modelo, não há um mestre. **Todos os nós são iguais (peers)**.

* Qualquer nó pode receber uma operação de escrita ou de leitura.
* Os nós se comunicam entre si para sincronizar os dados.

**Vantagens:**

* **Excelente Resiliência e Disponibilidade:** A falha de qualquer nó não impede que o sistema continue aceitando leituras e escritas, pois não há um ponto único de falha.

**Desvantagens:**

* **Complexidade de Consistência:** O maior desafio é o **conflito de escrita-escrita**. O que acontece se dois usuários tentarem atualizar o mesmo registro em dois nós diferentes ao mesmo tempo? O sistema precisa de mecanismos complexos para detectar e resolver esses conflitos, o que adiciona uma camada de complexidade significativa.

**Combinando Estratégias: Sharding com Replicação**

A arquitetura mais poderosa e comum em sistemas de larga escala é a **combinação de sharding com replicação mestre-escravo**.

Funciona assim:

1. O conjunto de dados é dividido em múltiplos **shards** (ex: Shard A, Shard B, Shard C).
2. **Cada shard é, ele mesmo, um mini-cluster com replicação mestre-escravo.**

* O Shard A tem seu próprio nó Mestre (que lida com as escritas do Shard A) e seus próprios nós Escravos.
* O Shard B tem outro nó Mestre (que lida com as escritas do Shard B) e seus próprios Escravos.

Essa abordagem oferece o melhor dos dois mundos:

* **Escalabilidade de Escrita:** As escritas são distribuídas entre os diferentes mestres de cada shard.
* **Escalabilidade de Leitura:** As leituras são distribuídas entre os escravos dentro de cada shard.
* **Alta Resiliência:** A falha do mestre de um shard afeta apenas aquele subconjunto de dados, e um de seus escravos pode assumir rapidamente, mantendo o sistema geral em funcionamento.

**Consistência**

Bancos de dados relacionais tentam exibir forte consistência evitando todas as várias inconsistências que discutiremos em breve. Uma vez que você começa a olhar para o mundo NoSQL, frases como "teorema CAP" e "consistência eventual" aparecem, e assim que você começa a construir algo, você tem que pensar sobre que tipo de consistência você precisa para seu sistema

Consistencia de atualização:

Abordagens para manter a consistência diante da simultaneidade são frequentemente descritas como pessimistas ou otimistas. Uma abordagem pessimista funciona prevenindo a ocorrência de conflitos; uma abordagem otimista permite que os conflitos ocorram, mas os detecta e toma medidas para resolvê-los.

Para conflitos de atualização, a abordagem pessimista mais comum é ter bloqueios de gravação, de modo que, para alterar um valor, você precisa adquirir um bloqueio, e o sistema garante que apenas um cliente possa obter um bloqueio por vez.

Uma abordagem otimista comum é uma atualização condicional onde qualquer cliente que faz uma atualização testa o valor logo antes de atualizá-lo para ver se ele mudou desde sua última leitura.

Há outra maneira otimista de lidar com um conflito de gravação-gravação: salvar as duas atualizações e registrar que elas estão em conflito. Essa abordagem é familiar para muitos programadores de sistemas de controle de versão, particularmente sistemas de controle de versão distribuídos que, por sua natureza, frequentemente terão confirmações conflitantes.

Abordagens pessimistas frequentemente degradam severamente a capacidade de resposta de um sistema a ponto de ele se tornar inadequado para seu propósito. Esse problema é agravado pelo perigo de erros — a simultaneidade pessimista frequentemente leva a deadlocks, que são difíceis de prevenir e depurar.

A replicação torna muito mais provável que ocorram conflitos de gravação-gravação. Se diferentes nós tiverem cópias diferentes de alguns dados que podem ser atualizados independentemente, você terá conflitos, a menos que tome medidas específicas para evitá-los.

Consistência de leitura

O período de tempo em que uma inconsistência está presente é chamado de janela de inconsistência.

consistência de replicação: garantir que o mesmo item de dados tenha o mesmo valor quando lido de réplicas diferentes

Embora a consistência da replicação seja independente da consistência lógica, a replicação pode exacerbar uma inconsistência lógica ao aumentar sua janela de inconsistência. Duas atualizações diferentes no mestre podem ser executadas em rápida sucessão, deixando uma janela de inconsistência de milissegundos. Mas atrasos na rede podem significar que a mesma janela de inconsistência dura muito mais em um escravo.

Existem algumas técnicas para fornecer consistência de sessão. Uma maneira comum, e geralmente a mais fácil, é ter uma sessão fixa: uma sessão que está vinculada a um nó (isso também é chamado de afinidade de sessão). Uma sessão fixa permite que você garanta que, enquanto você mantiver a consistência de leitura-suas-escritas em um nó, você a obterá para sessões também. A desvantagem é que as sessões fixas reduzem a capacidade do balanceador de carga de fazer seu trabalho.

Outra abordagem para consistência de sessão é usar carimbos de versão (“Carimbos de versão”, p. 61) e garantir que cada interação com o armazenamento de dados inclua o carimbo de versão mais recente visto por uma sessão. O nó do servidor deve então garantir que ele tenha as atualizações que incluem esse carimbo de versão antes de responder a uma solicitação.

Consistência é uma coisa boa — mas, infelizmente, às vezes temos que sacrificá-la. É sempre possível projetar um sistema para evitar inconsistências, mas muitas vezes é impossível fazer isso sem fazer sacrifícios insuportáveis em outras características do sistema. Como resultado, muitas vezes temos que trocar consistência por outra coisa.