Aula 27 – Estratégias de Cache – Parte 2: Cache Distribuído

No dinâmico universo do desenvolvimento de aplicações web, a busca por performance e escalabilidade é uma constante. Imagine que sua aplicação é uma loja online que, de repente, recebe milhares de visitantes simultaneamente. Se cada clique significar uma nova consulta ao banco de dados principal, o sistema rapidamente entrará em colapso, resultando em lentidão, erros e, pior, perda de clientes. É aqui que o cache entra em cena, atuando como um verdadeiro super-herói silencioso, guardando informações frequentemente acessadas para entregá-las com agilidade surpreendente.

Na aula anterior, exploramos os fundamentos do cache, entendendo como ele pode acelerar o acesso a dados e reduzir a carga sobre os sistemas de backend. No entanto, à medida que as aplicações crescem e se tornam mais complexas, distribuídas em múltiplos servidores ou microsserviços, o cache local, confinado a uma única instância, revela suas limitações. Precisamos de uma solução que acompanhe essa evolução, permitindo que todos os componentes da nossa arquitetura compartilhem informações em cache de forma eficiente e consistente.

Nesta aula, mergulharemos no fascinante mundo do **cache distribuído**. Nosso objetivo é que, ao final, você seja capaz de compreender o que é um cache distribuído, identificar os cenários ideais para sua aplicação, conhecer as principais tecnologias como Redis e Memcached, e dominar os padrões de interação com o cache, como Cache-Aside, Read-Through e Write-Through. Prepare-se para desvendar como essas estratégias podem transformar a performance e a resiliência de suas aplicações, tornando-as prontas para os desafios da web moderna.

Limitações do Cache Local

Por Que o Cache Local Não Basta?

Quando pensamos em otimização, o cache local, aquele que reside na memória de uma única instância da aplicação, é um excelente ponto de partida. Ele é rápido, fácil de implementar e reduz significativamente o tempo de resposta para dados que são acessados repetidamente por aquele servidor específico. Contudo, o mundo das aplicações web raramente se mantém em uma única instância por muito tempo, especialmente quando a demanda por escalabilidade e resiliência começa a crescer.

Problema: Imagine que sua aplicação, antes rodando em um único servidor, agora precisa lidar com um volume de tráfego que exige a adição de mais cinco servidores idênticos, todos atendendo aos mesmos usuários. Se cada um desses servidores tiver seu próprio cache local e independente, o que acontece?

Cada servidor precisará popular seu próprio cache, gerando múltiplas requisições ao banco de dados para os mesmos dados. Além disso, se um dado é atualizado, como garantir que todos os caches locais sejam invalidados ou atualizados de forma consistente? A complexidade e a ineficiência se tornam evidentes, e a promessa de performance do cache começa a se diluir.

> Pense em uma analogia simples: o cache local é como ter uma pequena geladeira em cada quarto de uma casa. Cada pessoa guarda suas próprias bebidas e lanches. Isso funciona bem para um consumo individual. Mas e se a família inteira precisar compartilhar um bolo grande ou um galão de suco? Seria muito mais eficiente ter uma despensa ou uma geladeira central na cozinha, acessível a todos, onde os itens compartilhados são armazenados e gerenciados de forma única. Essa é a essência da necessidade de um cache distribuído: um local centralizado e compartilhado para dados que beneficiam múltiplas instâncias da sua aplicação.

Desvendando o Cache Distribuído

Compreendendo as limitações do cache local em cenários de alta escalabilidade, a solução natural que emerge é o **cache distribuído**. Em sua essência, um cache distribuído é um sistema de armazenamento de dados em memória que é compartilhado por múltiplas instâncias de uma aplicação, ou até mesmo por diferentes aplicações. Ele atua como uma camada intermediária entre a aplicação e o banco de dados principal, armazenando cópias de dados frequentemente acessados para que possam ser recuperados rapidamente.

A grande sacada do cache distribuído é que ele não está atrelado a uma única máquina ou processo. Em vez disso, ele é composto por um cluster de servidores de cache que trabalham em conjunto, formando um pool de memória lógica. Quando uma instância da sua aplicação precisa de um dado, ela consulta esse pool de cache distribuído. Se o dado estiver lá, ele é retornado quase instantaneamente, sem a necessidade de acessar o banco de dados. Se não estiver, a aplicação busca no banco de dados, e então armazena uma cópia no cache distribuído para futuras requisições.

Escalabilidade

O cache pode ser expandido adicionando-se mais servidores ao cluster, distribuindo a carga de armazenamento e acesso.

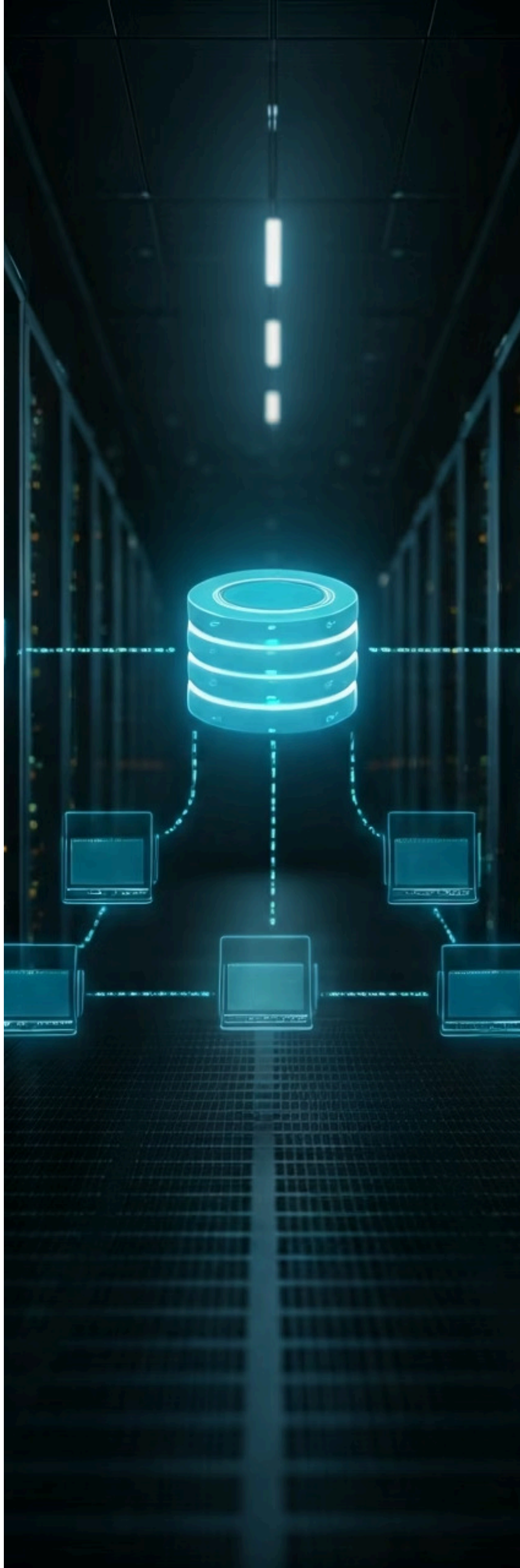
Resiliência

A falha de um único nó de cache não derruba todo o sistema.

Performance

A latência de acesso aos dados é drasticamente reduzida, liberando o banco de dados para operações mais complexas.

É como ter uma biblioteca central de referência rápida, acessível a todos os pesquisadores simultaneamente, em vez de cada um ter que ir à fonte original para cada consulta.



Cenários Ideais para o Cache Distribuído

Embora o cache distribuído seja uma ferramenta poderosa, ele não é uma solução mágica para todos os problemas de performance. Sua implementação deve ser estratégica, focada em cenários onde seus benefícios são maximizados e seus desafios gerenciados de forma eficaz. A chave é identificar os tipos de dados e padrões de acesso que mais se beneficiarão de uma camada de cache compartilhada.

1

Dados Frequentemente Lidos

Pense em um catálogo de produtos de um e-commerce, informações de perfil de usuário (nome, avatar), ou configurações globais da aplicação. Esses dados são acessados por inúmeras requisições, mas suas atualizações são esporádicas.

2

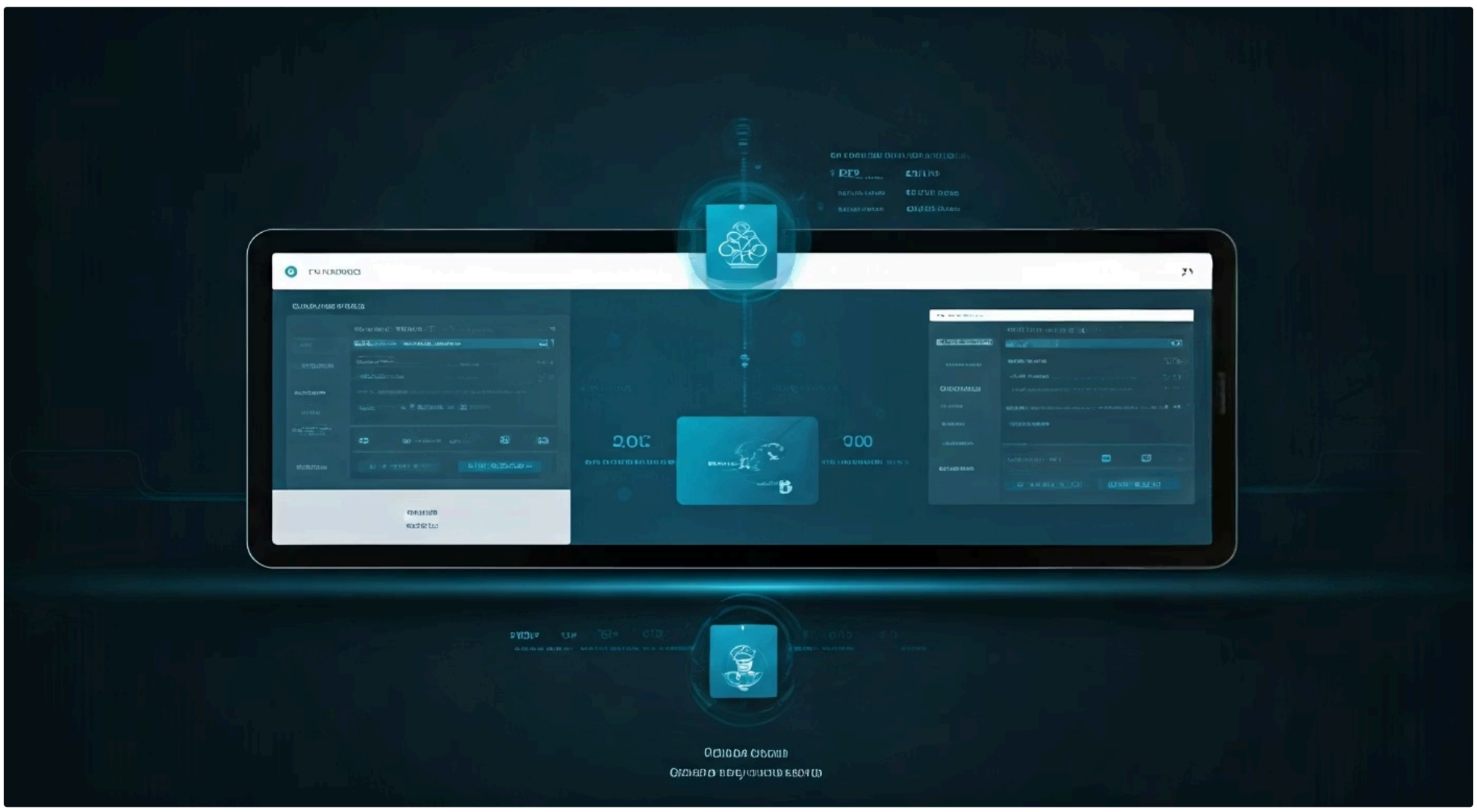
Sessões de Usuário

Em aplicações distribuídas, manter o estado da sessão em um cache compartilhado permite que o usuário seja roteado para qualquer instância da aplicação sem perder seu contexto, garantindo uma experiência fluida e sem interrupções.

3

Consultas Complexas

Se sua aplicação precisa gerar um relatório demorado ou calcular um score de recomendação que leva tempo, armazenar o resultado no cache evita que essa computação seja refeita a cada requisição.



> É como ter um "gabarito" para perguntas difíceis: uma vez que a resposta é encontrada, ela é guardada para ser consultada rapidamente por todos que precisarem. Ao analisar os gargalos de performance da sua aplicação, procure por esses padrões de acesso e considere o cache distribuído como uma solução robusta.

Redis: O Canivete Suíço do Cache Distribuído

Quando se fala em cache distribuído, é quase impossível não mencionar o **Redis**. O Remote Dictionary Server, como o nome sugere, é muito mais do que um simples cache de chave-valor; ele é um servidor de estruturas de dados em memória, versátil e extremamente rápido. Sua popularidade se deve à sua capacidade de lidar com uma vasta gama de casos de uso, indo muito além do cache tradicional.

O Redis armazena dados em memória, o que garante latências de acesso na ordem de microssegundos. Mas o que o diferencia é a riqueza de suas estruturas de dados. Além de strings simples (o clássico chave-valor), ele suporta listas, conjuntos (sets), conjuntos ordenados (sorted sets), hashes e até mesmo bitmaps e HyperLogLogs. Essa flexibilidade permite que o Redis seja usado para cache de objetos complexos, filas de mensagens (pub/sub), contadores em tempo real, tabelas de classificação (leaderboards), e até mesmo como um banco de dados primário para casos de uso específicos que exigem extrema velocidade.

Exemplo Prático

Um exemplo prático de sua versatilidade seria o cache de um feed de notícias personalizado. Em vez de apenas armazenar o HTML renderizado (uma string), você poderia usar uma lista do Redis para guardar os IDs das notícias de um usuário em ordem cronológica, e hashes para armazenar os detalhes de cada notícia. Quando o usuário acessa o feed, a aplicação rapidamente busca a lista de IDs e depois os detalhes das notícias no cache, tudo com pouquíssima latência.

O Redis também oferece recursos de persistência (opcional, para não perder dados em caso de reinício) e replicação, garantindo alta disponibilidade e durabilidade dos dados. É como ter um canivete suíço na sua caixa de ferramentas de desenvolvimento: uma única ferramenta com múltiplas funções, todas executadas com maestria.

Memcached: A Essência da Simplicidade para Cache

Enquanto o Redis se destaca pela sua versatilidade e riqueza de funcionalidades, o **Memcached** brilha pela sua simplicidade e foco singular: ser um cache de objetos em memória extremamente rápido. Ele é um sistema distribuído de cache de objetos de memória de código aberto e de alto desempenho, projetado para acelerar aplicações web dinâmicas, aliviando a carga sobre bancos de dados.

Filosofia

A filosofia do Memcached é "faça uma coisa e faça-a bem". Ele funciona como um grande hash table distribuído, onde você pode armazenar dados como pares de chave-valor. As chaves são strings e os valores podem ser qualquer tipo de dado serializável, como strings, números ou objetos JSON. Sua arquitetura é minimalista, sem suporte nativo para persistência de dados, replicação ou estruturas de dados avançadas como as do Redis. Isso significa que, em caso de reinício do servidor Memcached, os dados em cache são perdidos.

Vantagens

Essa simplicidade, no entanto, é sua maior força. O Memcached é incrivelmente eficiente e rápido para o que se propõe: cache de dados puros. Ele é ideal para cenários onde a performance bruta é a prioridade máxima e a perda de dados em cache é aceitável (pois os dados originais estão sempre no banco de dados). Por exemplo, cachear resultados de consultas SQL complexas que são frequentemente acessadas, mas não precisam de garantias de persistência.

> A aplicação tenta buscar o dado no Memcached; se não encontra, vai ao banco de dados, e então armazena o resultado no Memcached para a próxima vez. É como ter uma prateleira de "itens mais vendidos" em um supermercado: são fáceis de pegar, rápidos de reabastecer, e se acabarem, você sempre pode ir ao estoque principal, mesmo que demore um pouco mais.

Redis vs. Memcached: Escolhendo a Ferramenta Certa

A escolha entre Redis e Memcached é uma decisão comum no desenvolvimento de arquiteturas distribuídas. Ambas são excelentes ferramentas para cache em memória, mas suas filosofias e conjuntos de recursos as tornam mais adequadas para diferentes cenários. Entender suas distinções é crucial para tomar a decisão que melhor se alinha às necessidades do seu projeto.

Redis

O Redis, como vimos, é um servidor de estruturas de dados multifuncional. Ele oferece uma gama rica de tipos de dados, como listas, sets, hashes, além de strings. Possui suporte nativo para persistência (opcional), replicação (para alta disponibilidade e leitura escalável) e recursos avançados como Pub/Sub e transações. Isso o torna ideal para casos de uso que exigem mais do que um simples cache, como filas de mensagens, contadores em tempo real, sessões de usuário com dados complexos, e até mesmo como um banco de dados primário para dados voláteis. Sua complexidade é um pouco maior, mas a flexibilidade compensa.

Memcached

Por outro lado, o Memcached é a personificação da simplicidade e velocidade. Ele é um cache de chave-valor puro, focado exclusivamente em armazenar dados em memória para acesso rápido. Não oferece persistência, replicação nativa ou estruturas de dados complexas. Sua principal vantagem é a eficiência de memória e a performance bruta para operações de get/set simples. É a escolha perfeita quando você precisa de um cache leve, escalável horizontalmente e onde a perda de dados em cache é aceitável (pois o banco de dados é a fonte da verdade). Para cachear resultados de consultas de banco de dados ou objetos simples, o Memcached é imbatível em sua categoria.

Comparação Detalhada

Característica	Redis	Memcached
Tipo de Dados	Strings, Listas, Sets, Hashes, Sorted Sets	Strings (chave-valor simples)
Persistência	Sim (opcional)	Não
Replicação	Sim (primário/secundário)	Não (gerenciado pela aplicação)
Uso Típico	Cache complexo, filas, sessões, Pub/Sub	Cache simples de objetos, resultados SQL
Recursos Extras	Transações, Lua scripting, TTL avançado	Simplicidade, alta performance pura
Memória	Mais uso (devido a estruturas)	Menos uso (otimizado para simples)

Estratégias de Interação

Dominando os Padrões de Cache: Além do Básico

Ter um sistema de cache distribuído como Redis ou Memcached é um grande passo, mas a verdadeira maestria reside em como sua aplicação interage com ele. Não basta apenas "jogar" dados no cache; é preciso seguir padrões bem definidos para garantir que o cache seja eficaz, consistente e não introduza novos problemas, como dados desatualizados ou inconsistências. A forma como a aplicação lê e escreve dados no cache e no banco de dados define a estratégia de cache.

> Pense na sua despensa compartilhada (o cache distribuído) e na sua fonte de ingredientes frescos (o banco de dados). Como você decide o que pegar da despensa e o que comprar no mercado? E quando você compra algo novo, onde você guarda? Essas decisões impactam a frescura dos seus ingredientes e a eficiência da sua cozinha. Da mesma forma, os padrões de cache são as "receitas" para gerenciar essa interação, garantindo que sua aplicação sempre sirva os dados mais frescos e rapidamente.

01

Cache-Aside

A aplicação gerencia o cache

02

Read-Through

O cache como proxy inteligente

03

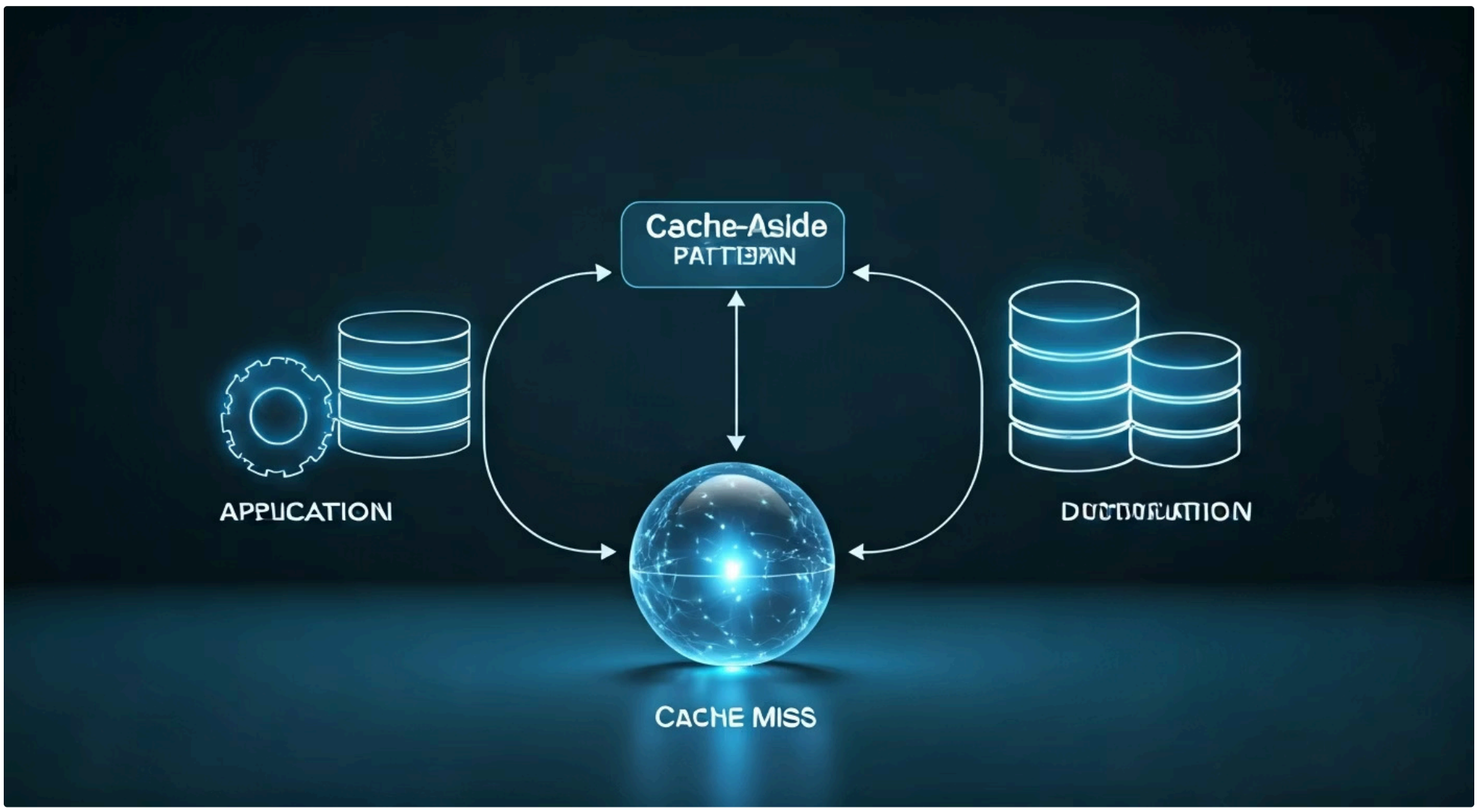
Write-Through

Garantindo consistência na escrita

Existem três padrões principais que dominam a interação com caches distribuídos: **Cache-Aside**, **Read-Through** e **Write-Through**. Cada um deles oferece uma abordagem diferente para lidar com a leitura e escrita de dados, com suas próprias vantagens e desvantagens em termos de complexidade, consistência e performance. Compreendê-los é fundamental para projetar sistemas robustos e eficientes. Vamos explorar cada um deles em detalhes, entendendo como eles funcionam e quando aplicá-los.

Cache-Aside: Onde a Aplicação Gerencia o Cache

O padrão **Cache-Aside**, também conhecido como "Lazy Loading", é talvez o mais comum e intuitivo entre as estratégias de cache. Nele, a lógica de interação com o cache e o banco de dados reside primariamente na própria aplicação. A aplicação age como o "gerente" que decide onde buscar os dados e onde armazená-los.



Como Funciona

A dinâmica é simples: quando a aplicação precisa de um dado, ela primeiro tenta buscá-lo no cache. Se o dado for encontrado (um "cache hit"), ele é retornado imediatamente. Isso é rápido e eficiente. No entanto, se o dado não estiver no cache (um "cache miss"), a aplicação então busca o dado no banco de dados principal. Uma vez recuperado do banco, o dado é armazenado no cache para futuras requisições e, em seguida, retornado à aplicação. Essa abordagem garante que apenas os dados que são realmente solicitados e usados sejam carregados no cache, economizando recursos.

Vantagens

- Simplicidade de implementação
- Controle total pela aplicação
- Flexibilidade na lógica de cache

Desafios

- Cache frio inicial
- Consistência gerenciada manualmente
- Invalidação explícita necessária

> Imagine que você vai à despensa (cache) para pegar um ingrediente. Se não tem, você vai ao mercado (banco de dados), compra, guarda na despensa e usa.

Read-Through: O Cache Como Proxy Inteligente

O padrão **Read-Through** é uma variação do Cache-Aside, mas com uma diferença fundamental: a lógica de busca no banco de dados em caso de cache miss é encapsulada dentro do próprio sistema de cache. Para a aplicação, o cache se comporta como um proxy inteligente, sempre fornecendo os dados, sem que ela precise se preocupar de onde eles vêm.

Funcionamento

Neste padrão, a aplicação sempre faz a requisição de dados diretamente ao cache. Se o cache já possui o dado (cache hit), ele o retorna imediatamente, assim como no Cache-Aside. A diferença surge no cenário de um cache miss: em vez de a aplicação ir ao banco de dados, é o próprio sistema de cache que se encarrega de buscar o dado na fonte de dados primária (o banco de dados). Uma vez recuperado, o cache armazena esse dado internamente e o retorna à aplicação.

Benefícios

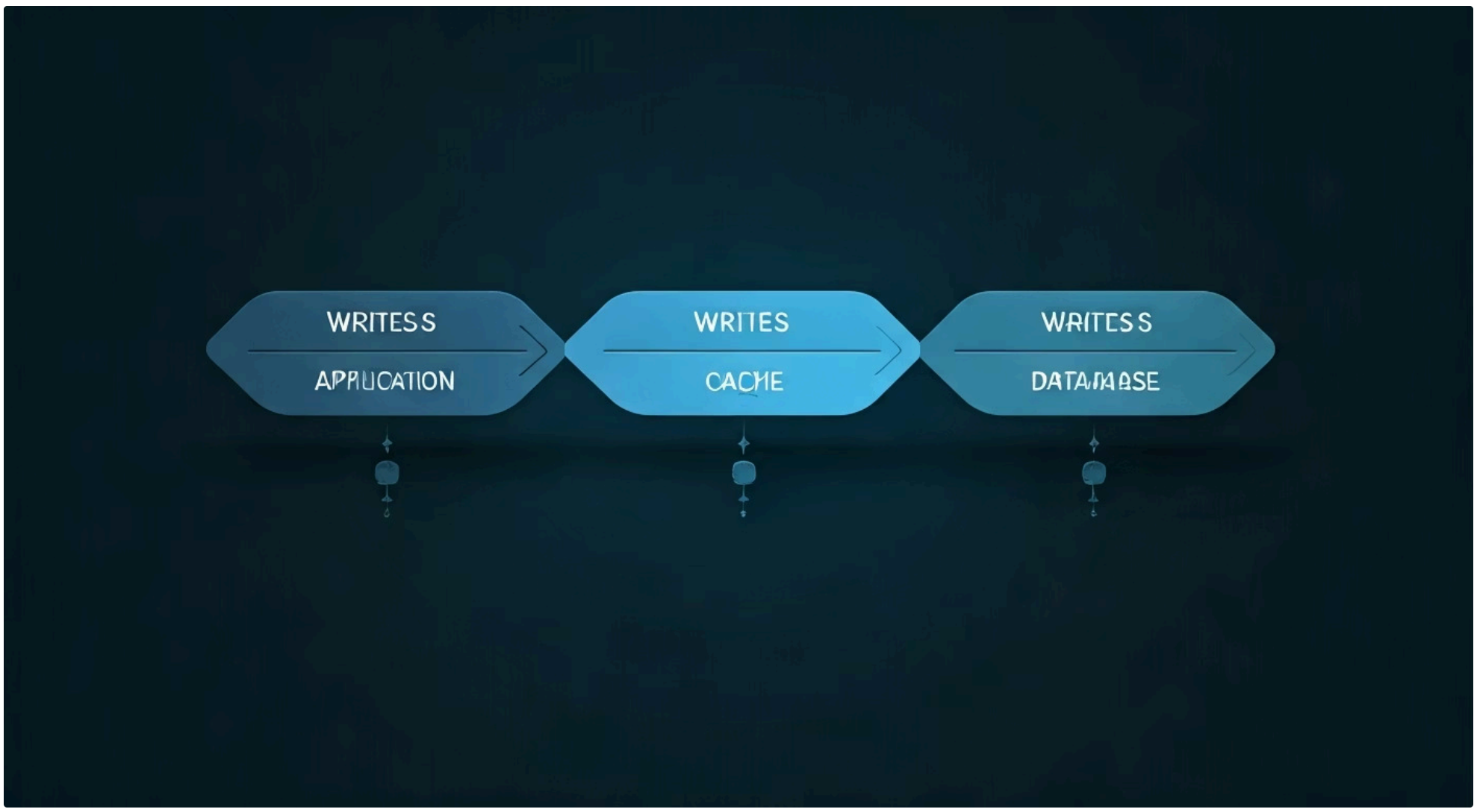
A principal vantagem do Read-Through é que ele simplifica a lógica da aplicação. O código da aplicação não precisa saber como buscar os dados do banco de dados em caso de cache miss; ele apenas interage com o cache. Isso pode levar a um código mais limpo e modular. No entanto, essa abstração adiciona complexidade ao próprio sistema de cache, que agora precisa ter a inteligência para se comunicar com o banco de dados. Além disso, se o cache for um ponto único de falha ou tiver problemas de performance ao buscar no banco, isso pode impactar toda a aplicação.

Analogia

É como ter um assistente pessoal que, se você pede algo e ele não tem à mão (cache), ele mesmo vai buscar para você (banco de dados) e te entrega, sem que você precise se preocupar com a logística.

Write-Through: Garantindo Consistência na Escrita

Enquanto os padrões Cache-Aside e Read-Through focam na otimização da leitura, o padrão **Write-Through** aborda a questão da escrita de dados, com um forte foco na consistência imediata entre o cache e o banco de dados. Ele é projetado para garantir que, sempre que um dado é modificado, essa modificação seja refletida tanto no cache quanto na fonte de dados primária.



Mecanismo

No padrão Write-Through, quando a aplicação precisa escrever ou atualizar um dado, ela o faz diretamente no cache. O sistema de cache, por sua vez, não apenas armazena o dado em sua memória, mas também o escreve **simultaneamente** no banco de dados principal. Somente após a escrita ser confirmada tanto no cache quanto no banco de dados é que a operação é considerada completa e um sucesso é retornado à aplicação.

Vantagem

Garantia de consistência imediata entre cache e banco de dados

Desvantagem

Maior latência de escrita devido à operação dupla

> É como quando você anota algo importante: você escreve no seu caderno (cache) e imediatamente faz uma cópia no seu arquivo principal (banco de dados) antes de considerar a tarefa concluída, garantindo que a informação esteja em ambos os lugares.

Navegando Pelos Desafios do Cache Distribuído

A implementação de um cache distribuído, embora traga imensos benefícios, não está isenta de desafios. É fundamental estar ciente dessas armadilhas para projetar e manter um sistema de cache robusto e eficiente. Ignorar esses pontos pode levar a problemas de consistência, performance inesperada e até mesmo a falhas na aplicação.

Invalidação de Cache

Como garantir que os dados no cache estejam sempre atualizados em relação à fonte da verdade (o banco de dados)? Se um dado é modificado no banco, mas o cache continua servindo a versão antiga, teremos inconsistência. Isso é conhecido como "stale data" (dados obsoletos).

Consistência de Dados

Em sistemas distribuídos, alcançar consistência forte (todos os nós veem a mesma informação ao mesmo tempo) é complexo e caro. Muitas vezes, opta-se por consistência eventual, onde os dados eventualmente se propagarão para todos os nós, mas pode haver um pequeno atraso.

Cache Stampede

Isso ocorre quando um item popular expira do cache e, de repente, milhares de requisições simultâneas tentam acessá-lo. Todas essas requisições resultam em um cache miss e, consequentemente, sobrecarregam o banco de dados, que tenta gerar o mesmo dado repetidamente.

Gerenciamento do Cluster

A complexidade de gerenciar um cluster de cache, monitorar sua saúde e garantir sua alta disponibilidade também são considerações importantes.

> É como gerenciar uma grande despensa compartilhada: você precisa de regras claras para saber quando um item está vencido, quem pode reabastecer e como evitar que todos corram para pegar o último item ao mesmo tempo.

Mantendo o Cache Fresco: Estratégias de Invalidação

A invalidação de cache é, sem dúvida, um dos aspectos mais complexos e críticos ao trabalhar com caches distribuídos. Um cache que serve dados obsoletos é pior do que não ter cache algum, pois pode levar a decisões erradas e inconsistências graves na aplicação. A arte da invalidação reside em encontrar o equilíbrio entre manter os dados frescos e evitar a sobrecarga do sistema.

Time-To-Live (TTL)

A estratégia mais básica é o **Time-To-Live (TTL)**. Cada item no cache recebe um tempo de vida. Após esse período, o item é automaticamente expirado e removido do cache. Na próxima requisição, ele será um cache miss e buscará o dado fresco do banco de dados. O TTL é simples de implementar e eficaz para dados que podem tolerar um certo grau de obsolescência ou que são atualizados em intervalos previsíveis. No entanto, para dados que mudam de forma imprevisível, um TTL fixo pode ser insuficiente.

Invalidação por Eventos

Para dados que exigem maior frescor, podemos usar a **invalidação baseada em eventos**. Sempre que um dado é modificado no banco de dados, um evento é disparado (por exemplo, via um sistema de mensagens como Kafka ou RabbitMQ) que instrui o cache a invalidar ou atualizar o item correspondente. Essa abordagem garante que o cache seja atualizado quase em tempo real.

Invalidação Manual

Outra estratégia é a **invalidação manual ou programática**, onde a aplicação explicitamente remove um item do cache após uma operação de escrita no banco de dados. Por exemplo, após um usuário atualizar seu perfil, a aplicação remove o perfil antigo do cache.

> É como gerenciar a validade dos produtos na sua despensa: alguns têm uma data de validade clara (TTL), outros você joga fora assim que usa (invalidação manual), e para outros, você recebe um aviso quando o produto é substituído por um novo lote (invalidação por evento).

Cache Distribuído no Mundo de Microserviços e Serverless

As arquiteturas modernas, dominadas por **microsserviços** e **serverless**, intensificaram a necessidade e a relevância do cache distribuído. Em um ambiente onde as aplicações são decompostas em pequenos serviços independentes, a comunicação entre eles e o acesso a dados compartilhados podem se tornar gargalos significativos. O cache distribuído surge como um pilar fundamental para manter a performance e a resiliência nesses ecossistemas complexos.

Microsserviços

Em uma arquitetura de microsserviços, cada serviço pode ter seu próprio banco de dados, mas muitos dados são compartilhados ou acessados por múltiplos serviços (por exemplo, dados de usuário, configurações). Um cache distribuído centralizado permite que esses serviços compartilhem dados em memória de forma eficiente, reduzindo a latência e a carga sobre os bancos de dados individuais. Ele também é crucial para o gerenciamento de sessões de usuário em um ambiente onde as requisições podem ser roteadas para qualquer instância de qualquer serviço.

Serverless

No contexto **serverless**, onde as funções são efêmeras e escalam sob demanda, o cache distribuído (muitas vezes oferecido como um serviço gerenciado na nuvem, como AWS ElastiCache para Redis ou Azure Cache for Redis) é ainda mais vital. Ele permite que as funções serverless, que não mantêm estado entre invocações, acessem dados rapidamente sem ter que re-consultar um banco de dados a cada vez. Além disso, tecnologias de comunicação modernas como **GraphQL** e **gRPC** podem se beneficiar enormemente do cache. Uma query GraphQL complexa, por exemplo, pode ser cacheada para evitar reprocessamento no backend, enquanto respostas de serviços gRPC podem ser armazenadas para acelerar interações subsequentes, otimizando a comunicação entre os serviços e o front-end. O cache distribuído é, portanto, um componente indispensável para construir sistemas ágeis, escaláveis e de alta performance na vanguarda do desenvolvimento web.

Recapitulando

Conclusão e Próximos Passos

Chegamos ao final de nossa jornada pela Parte 2 das Estratégias de Cache, focando no poderoso conceito de cache distribuído. Vimos que, em um mundo de aplicações web cada vez mais escaláveis e complexas, o cache local rapidamente atinge seus limites. O cache distribuído surge como a solução robusta, oferecendo um pool de memória compartilhado que acelera o acesso a dados, reduz a carga sobre os bancos de dados e aumenta a resiliência de todo o sistema.

Ferramentas

Exploramos as ferramentas mais populares, Redis e Memcached, entendendo suas forças e quando cada uma é a escolha ideal.

Padrões

Desvendamos os padrões de interação com o cache – Cache-Aside, Read-Through e Write-Through – compreendendo como cada um gerencia a leitura e escrita de dados.

Desafios

Discutimos os desafios inerentes ao cache distribuído, como a invalidação e a consistência.

Integração

Vimos como ele se integra perfeitamente às arquiteturas modernas de microsserviços e serverless.

Em prática

Comece identificando os dados mais acessados e menos modificados em sua aplicação. Escolha a tecnologia de cache distribuído que melhor se alinha às suas necessidades (Redis para complexidade, Memcached para simplicidade). Implemente um padrão de cache, começando pelo Cache-Aside, e não subestime a importância de uma estratégia de invalidação bem definida. Monitore a performance e a consistência para refinar sua abordagem.

Autoavaliação

1

Questão 1

Qual das seguintes afirmações melhor descreve a principal vantagem do cache distribuído em relação ao cache local em uma arquitetura de microsserviços?

- a) Reduz a complexidade do código da aplicação.
- b) Permite que múltiplas instâncias da aplicação compartilhem dados em cache de forma consistente.
- c) Elimina completamente a necessidade de um banco de dados.
- d) Garante que todos os dados em cache sejam sempre 100% atualizados.

2

Questão 2

Um desenvolvedor precisa de um cache distribuído que suporte estruturas de dados complexas como listas e hashes, além de oferecer persistência opcional e replicação. Qual tecnologia seria a mais indicada?

- a) Memcached
- b) MySQL
- c) Redis
- d) Apache Cassandra

3

Questão 3

No padrão Cache-Aside, qual é a sequência correta de operações quando a aplicação precisa de um dado e ele não está no cache (cache miss)?

- a) Aplicação busca no banco de dados, armazena no cache, retorna o dado.
- b) Cache busca no banco de dados, armazena em si mesmo, retorna o dado.
- c) Aplicação armazena no cache, depois busca no banco de dados, retorna o dado.
- d) Aplicação busca no cache, se não encontra, retorna erro.

4

Questão 4

Qual dos padrões de cache é mais adequado para cenários onde a consistência imediata entre o cache e o banco de dados é crítica para operações de escrita, mesmo que isso aumente ligeiramente a latência da escrita?

- a) Cache-Aside
- b) Read-Through
- c) Write-Through
- d) Write-Back

Gabarito

1. b) **2.** c) **3.** a) **4.** c)

Questão Discursiva

> Explique a importância da invalidação de cache em sistemas distribuídos e descreva duas estratégias comuns para lidar com ela, abordando suas vantagens e desvantagens.

Continue Aprendendo

Próxima Aula e Recursos

Próxima Aula

Aula 28 – Otimização de Front-End. Nesta aula, exploraremos como otimizar a parte visível da aplicação, garantindo que a experiência do usuário seja tão rápida e fluida quanto o backend que acabamos de otimizar.

Recursos Adicionais

- **Documentação Oficial do Redis**: Para aprofundar nas funcionalidades e comandos do Redis.
- **Documentação Oficial do Memcached**: Para entender a arquitetura e uso do Memcached.
- **Artigos sobre Padrões de Cache (Martin Fowler)**: Para uma visão mais conceitual e arquitetural dos padrões.

NOTA IMPORTANTE: As informações técnicas desta aula estão atualizadas até 2025. Consulte sempre as documentações oficiais das tecnologias e as melhores práticas da indústria para verificar alterações e novas funcionalidades.