Apache Kafka é uma plataforma de streaming de eventos usada para coletar, armazenar e processar fluxos de dados em tempo real, em grande escala. Ela tem inúmeros casos de uso, incluindo registro distribuído, processamento de streams e mensageria Pub-Sub.

Kafka armazena eventos, como os eventos entram e saem, como analisá-los, todas essas coisas.

Evento é apenas uma coisa que aconteceu, é isso.

Um evento pode ser a mudança no status de algum processo de negócios. Um evento pode ser algum tipo de interação do usuário. Um microserviço completa alguma unidade de trabalho e quer colocar o registro dessa unidade de trabalho em algum lugar. Isso é um evento. Todas essas coisas são eventos.

Elas são apenas coisas que aconteceram combinadas com a descrição do que aconteceu. Então, um evento é uma combinação de notificação e Status

1. **Eventos** no Kafka: São modelados como pares chave/valor, onde tanto a chave quanto o valor são sequências de bytes. Internamente, o Kafka trata esses dados como bytes sem tipo específico, mas externamente, em linguagens de programação, os dados são geralmente estruturados (tipados)​.
2. **Serialização e Desserialização**: Esses processos convertem os dados estruturados das linguagens de programação (objetos, strings, etc.) para bytes (serialização) e vice-versa (desserialização). Formatos comuns para essa conversão incluem JSON, Avro, e Protobuf​.
3. **Estrutura dos Dados:** O valor de um evento Kafka geralmente representa um objeto de domínio da aplicação, como dados de um sensor. Essa estrutura é importante para que os dados possam ser corretamente interpretados pela aplicação​.
4. **Chaves no Kafka:** As chaves podem ser primitivas (como strings ou inteiros) ou objetos complexos serializados. Elas não são necessariamente identificadores únicos dos eventos, mas frequentemente identificam entidades no sistema, como um usuário ou dispositivo específico​ .

O Kafka é uma coleção de logs imutáveis e apendáveis, sendo que o principal componente de armazenamento é o **tópico**, que representa uma entidade de dados específica.

Tópicos são quebrados em componentes menores chamados **partições.** Quando você escreve uma mensagem para um tópico Kafka, ela é armazenada em uma das partições do tópico, determinada pela chave da mensagem.

Confluent Cloud

Criar Tópico:Na página inicial selecione a aba "Topics" à esquerda da tela e escolha "Create topic".

Criar mensagem:

1. Pelo console: Selecione "Messages" para ver a tela de visualização de mensagens. Selecione "Produce a new message to this topic". Isso abrirá uma interface onde você pode inserir uma chave e um valor para uma nova mensagem. Clique em "Produce" para adicionar a mensagem ao tópico.

Outros métodos para produzir dados em um tópico Kafka:

1. API
2. Kafka Connect
3. interface de linha de comando (CLI) do Confluent Cloud: No canto inferior esquerdo da tela, você verá a aba "CLI and tools". Siga as instruções para baixar e atualizar as ferramentas de linha de comando.

No terminal, execute confluent login -save e insira suas credenciais do Confluent Cloud. Use a flag "save" para salvar suas credenciais localmente.

Em seguida, selecione o ambiente e um cluster. Execute confluent environment list para ver uma lista de ambientes. Use o ambiente padrão e anote seu ID. Execute confluent environment use [ID]. Depois, execute confluent kafka cluster list e anote o ID do cluster. Execute confluent kafka cluster use [ID]. Para conectar ao cluster, crie uma chave de API com confluent api-key create e use confluent api-key use [key]. Agora, você pode começar a interagir com o cluster.

**Tópico** unidade fundamental de organização de eventos

Você cria diferentes tópicos, geralmente para armazenar diferentes tipos de eventos, e pode criar diferentes tópicos para armazenar versões filtradas e transformadas do mesmo tipo de evento.

Um tópico é um log de eventos

Primeiro de tudo, eles são apenas de anexação, portanto. quando você escreve uma nova mensagem em um log, ela sempre é adicionada ao final do log.

Segundo, ler dados em um log do Kafka implica iniciar a leitura em um ponto específico (o deslocamento) e continuar lendo sequencialmente. Isso é diferente de sistemas que usam índices, onde você pode pular diretamente para qualquer posição específica sem a necessidade de leitura sequencial.

Terceiro, os eventos em um log são imutáveis. eventos em tópicos do Kafka são imutáveis, Uma vez que um evento é registrado no log, ele não pode ser alterado ou removido, , uma vez que um evento é registrado, ele permanece lá sem alterações.

A imutabilidade dos eventos facilita para o Kafka manter uma alta taxa de transferência (rapidez e volume de processamento de eventos) dentro e fora dos tópicos.

No Kafka, você pode configurar tópicos para que as mensagens expirem após um certo tempo ou quando atingem um certo tamanho.

O período de retenção pode variar desde alguns segundos até vários anos, ou até mesmo ser configurado como infinito.

e esses dados podem viver para sempre, e os logs que sustentam esses tópicos do Kafka são arquivos armazenados em disco. Então, quando você escreve um evento para um tópico, ele é tão durável quanto seria se você tivesse escrito no sistema de arquivos ou em qualquer banco de dados

O Kafka pode ter um desempenho impressionante mesmo em um único servidor, o Kafka é um sistema distribuído, então o foco geralmente não é apenas no desempenho de um nó, mas no sistema como um todo.

**Partitions:** as partições são úteis para permitir que você divida seu tópico em partes mais gerenciáveis que podem ser armazenadas em vários nós no seu cluster.

Então todas as vezes que você criar um tópico, pense em quantas partitions vc precisará para ter um bom desempenho no cluster.

Brokers: Do ponto de vista da infraestrutura física, o Kafka é composto por uma rede de máquinas chamadas brokers . Esses brokers podem não ser servidores físicos separados; eles podem ser instâncias na nuvem, contêineres em pods, ou servidores virtualizados em data centers físicos. Independentemente da implantação, seja na nuvem ou em servidores físicos, são máquinas independentes, cada uma executando o processo do broker Kafka.

Cada broker hospeda um conjunto de partições do Kafka e lida com solicitações para gravar ou ler eventos, além de gerenciar a replicação de partições entre si. Fora isso, são mantidos intencionalmente simples para facilitar o escalonamento e a manutenção.

Replicação: Armazenar cada partição em apenas um broker não é uma boa prática. Brokers, sejam servidores físicos ou contêineres gerenciados, estão sujeitos a falhas em seu armazenamento subjacente. Para proteger os dados, é necessário copiar as partições para vários outros brokers. Essas cópias são chamadas de **réplicas seguidoras**, enquanto a partição principal é chamada de **réplica líder.**

Cada partição replicada possui um líder e N-1 seguidores. Ao produzir dados para a partição, você está realmente produzindo para o líder. Em geral, tanto a escrita quanto a leitura de dados de uma partição envolvem o líder. Após a escrita, o líder e os seguidores trabalham juntos para replicar os novos dados, garantindo que não fiquem apenas no líder.

Como desenvolvedor, você não precisa se preocupar com a replicação ao usar o Kafka. A replicação ocorre automaticamente e você só precisa saber que ela está acontecendo. No entanto, se você estiver administrando um cluster Kafka, pode ser necessário ajustar configurações de replicação com frequência, ou contar com um serviço totalmente gerenciado como o Confluent Cloud.

Desde que a replicação esteja ativada, que é o padrão, você escreve para o líder e os seguidores obtêm os novos dados automaticamente. Existem configurações de gravação que podem ser ajustadas para garantir diferentes níveis de durabilidade, portanto se um nó no cluster falhar, outro assumirá seu papel e as mensagens e o tópico permanecerão acessíveis.

Aplicações que usam Kafka: **produtores** e **consumidores.**

Eles colocam mensagens em tópicos e leem mensagens dos tópicos. Todo componente da plataforma Kafka que não é um broker Kafka é, no fundo, um produtor, um consumidor ou ambos.

Em Java, que é a linguagem nativa do Apache Kafka, você usa a classe **KafkaProducer** para se conectar ao cluster. Aqui está um guia passo a passo para entender como isso funciona:

1. Criação do KafkaProducer:
   * Classe KafkaProducer: Esta é a classe principal que você utilizará para produzir mensagens para o cluster Kafka. Para inicializá-la, você precisa fornecer um mapa de parâmetros de configuração.
2. Configuração do KafkaProducer:
   * Parâmetros de Configuração: Estes parâmetros incluem informações importantes como:
     + Endereços dos Brokers: Você precisa fornecer os endereços de alguns brokers no cluster. Não é necessário listar todos os brokers; apenas dois ou três são suficientes para começar.
     + Configurações de Segurança: Se o seu cluster usa autenticação e criptografia, você precisará incluir as configurações apropriadas.
     + Outras Configurações de Rede: Isso pode incluir ajustes no comportamento da rede, como o tamanho do buffer e os tempos de espera.
3. Criação do ProducerRecord:
   * Classe ProducerRecord: Esta classe é usada para encapsular os eventos que você deseja enviar para o cluster.
   * Par Chave-Valor: Os eventos no Kafka são modelados como pares chave-valor, onde a chave pode ser usada para determinar a partição para a qual a mensagem será enviada, e o valor é o próprio conteúdo da mensagem. ProducerRecord é a classe que você usa para criar essas mensagens.

Aqui está um exemplo simplificado de como você pode configurar um KafkaProducer e enviar uma mensagem:

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class KafkaProducerExample {

public static void main(String[] args) {

// Configurações do produtor

Properties props = new Properties();

props.put("bootstrap.servers", "broker1:9092,broker2:9092"); // Endereços dos brokers

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

// Criação do KafkaProducer

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

// Criação do ProducerRecord

ProducerRecord<String, String> record = new ProducerRecord<>("meu-topico", "minha-chave", "meu-valor");

// Envio da mensagem

producer.send(record);

// Fechamento do produtor

producer.close();

}

}

Neste exemplo:

* Propriedades do Produtor: Configurações essenciais como os endereços dos brokers (bootstrap.servers) e os serializadores para a chave e o valor.
* KafkaProducer: Inicializa o produtor com as configurações fornecidas.
* ProducerRecord: Cria uma mensagem com um tópico (meu-topico), uma chave (minha-chave) e um valor (meu-valor).
* Envio da Mensagem: Usa o método send do KafkaProducer para enviar a mensagem para o cluster.
* Fechamento do Produtor: Chama o método close para liberar os recursos usados pelo produtor.

É o produtor que decide para qual partição enviar cada mensagem. Aqui estão as principais formas de fazer isso:

1. Round-Robin para Mensagens Sem Chave: Se a mensagem não tem uma chave, o produtor pode distribuir as mensagens de forma sequencial entre as partições. Isso ajuda a balancear a carga uniformemente.
2. Cálculo de Partição Usando Hash da Chave: Hash da Chave: Se a mensagem tem uma chave, o produtor calcula a partição de destino fazendo um hash da chave. Isso garante que mensagens com a mesma chave sempre vão para a mesma partição, o que é útil para garantir a ordem das mensagens.
3. Esquema Personalizado: Esquema Personalizado: Embora não seja comum, você pode configurar um esquema personalizado para determinar a partição de destino. Isso permite maior controle sobre a distribuição das mensagens, dependendo de necessidades específicas da aplicação.

Portanto, o particionamento é uma responsabilidade do produtor no Kafka.

Conectando ao Cluster:

Em Java, você usa a classe KafkaConsumer para se conectar ao cluster. Você precisa passar um mapa de parâmetros de configuração, que inclui:

* Endereços dos Brokers: Especifique alguns brokers no cluster.
* Configurações de Segurança: Inclua qualquer parâmetro de segurança necessário.
* Outras Configurações: Ajustes de rede e outras configurações.

Subscrição a Tópicos:

Com a conexão estabelecida, você pode subscrever a um ou mais tópicos. Isso pode ser feito especificando uma lista de tópicos ou usando uma expressão regular para combinar com tópicos.

Recebendo Mensagens:

Quando há mensagens disponíveis, elas são recebidas como uma coleção chamada ConsumerRecords, que contém instâncias de ConsumerRecord. Cada ConsumerRecord é um par chave-valor representando uma única mensagem.

Gerenciamento Interno:

O KafkaConsumer gerencia:

* Pools de Conexão: Mantém as conexões com o cluster.
* Metadados do Cluster: Atualiza-se com informações sobre falhas de nós, reatribuição de partições, etc.
* Protocolo de Rede: Similar ao KafkaProducer, gerencia toda a comunicação com o cluster.

Leitura sem Destruição:

Diferente de sistemas de filas de mensagens tradicionais, ler uma mensagem no Kafka não a remove do tópico. Isso permite que múltiplos consumidores leiam do mesmo tópico sem interferirem uns nos outros.

Escalabilidade dos Consumidores:

Se uma única instância do seu aplicativo consumidor não é suficiente para processar todas as mensagens devido ao volume ou complexidade do processamento, você pode escalar facilmente:

* Grupos de Consumidores: Ao adicionar mais instâncias do consumidor com o mesmo group.id, o Kafka redistribui as partições automaticamente entre as instâncias.
* Rebalanceamento Automático: Cada vez que você adiciona ou remove uma instância, o Kafka reequilibra as partições entre os consumidores disponíveis.

Garantia de Ordem:

* Ordenação por Partição: Mensagens dentro de uma partição são sempre processadas em ordem.
* Escalabilidade com Ordenação: Você pode escalar até o número de partições disponíveis. Se houver mais instâncias do que partições, algumas instâncias ficarão ociosas.

Contraste com Sistemas Tradicionais:

Em sistemas tradicionais de mensagens:

* Filas de Mensagens: Permitem escalar consumidores, mas podem perder a ordenação das mensagens.
* Tópicos de Mensagens: Mantêm a ordenação, mas sacrificam a capacidade de escalar consumidores.
* No Kafka, grupos de consumidores permitem escalar horizontalmente mantendo a melhor garantia de ordenação possível.

**Ferramentas adicionais**

Resolvem problemas comuns de maneira padronizada e são amplamente utilizadas e testadas pela comunidade.

* Kafka Connect: Facilita a integração de Kafka com outros sistemas de dados, fornecendo uma maneira padronizada de mover grandes volumes de dados entre Kafka e outros sistemas.
* Confluent Schema Registry: Gerencia esquemas de dados para Kafka, garantindo que produtores e consumidores possam entender e validar os dados que estão enviando e recebendo.
* Kafka Streams: Uma biblioteca para construir aplicativos de processamento de stream, permitindo que você processe dados em tempo real diretamente dentro do Kafka.
* ksqlDB: Permite realizar consultas SQL diretamente em dados de streams no Kafka, simplificando o processamento e análise de dados em tempo real.

Explorando o **Confluent Schema Registry**: Facilitando a Evolução de Esquemas no Kafka

Quando aplicações estão produzindo e consumindo mensagens no Kafka, dois cenários comuns surgem:

1. Novos Consumidores: Novas aplicações, possivelmente desenvolvidas por diferentes equipes ou pessoas, começam a consumir mensagens de tópicos existentes. Estes novos consumidores precisam entender o formato das mensagens nesses tópicos.
2. Evolução do Formato das Mensagens: O formato das mensagens evolui conforme o negócio muda. Por exemplo, um objeto de pedido pode ganhar novos campos de status ou o nome do usuário pode ser dividido em nome e sobrenome.

Para lidar com essas mudanças de esquema, existe o Confluent Schema Registry.

O Que é o Schema Registry?

O Schema Registry é um processo servidor independente que mantém um banco de dados de todos os esquemas utilizados nos tópicos do Kafka. Este banco de dados é armazenado em um tópico interno do Kafka e é cacheado no Schema Registry para acesso rápido.

Principais Funções:

* Validação de Esquemas: Permite que produtores e consumidores verifiquem se as mensagens que estão produzindo ou consumindo são compatíveis com versões anteriores dos esquemas.
* Gerenciamento de Esquemas: Ajuda a gerenciar a evolução dos esquemas, prevenindo falhas em tempo de execução devido a incompatibilidades de esquema.

Como Funciona?

1. Configuração do Produtor: Quando um produtor é configurado para usar o Schema Registry, ele chama uma API REST do Schema Registry no momento da produção de uma mensagem. Ele apresenta o esquema da nova mensagem:
   * Se for igual ao último esquema produzido, a produção será bem-sucedida.
   * Se for diferente, mas compatível com as regras definidas, a produção ainda pode ser bem-sucedida.
   * Se for incompatível, a produção falhará, e a aplicação será notificada.
2. Configuração do Consumidor: Se um consumidor tentar ler uma mensagem com um esquema incompatível, o Schema Registry informará que a mensagem não deve ser consumida.

Formatos de Serialização Suportados:

* JSON Schema
* Avro
* Protobuf

Ferramentas e Integração:

O Schema Registry suporta IDLs (Interface Description Languages) que permitem descrever o esquema dos objetos em arquivos de texto que podem ser controlados por versão. Por exemplo, usando Avro, você pode criar um arquivo .avsc, descrevendo o esquema de um objeto, que pode ser transformado em um objeto Java usando plugins do Maven ou Gradle.

Vantagens do Schema Registry:

* Redução de Falhas em Tempo de Execução: Evita falhas devido a mudanças de esquema incompatíveis.
* Colaboração Facilitada: Permite a colaboração em mudanças de esquema através de processos padrão de controle de versão, como pull requests.
* Compatibilidade de Esquemas: Ferramentas que verificam compatibilidade de esquemas em tempo de build e runtime.

Conclusão:

O uso do Schema Registry é essencial em sistemas não triviais, pois ajuda a gerenciar a evolução dos esquemas de forma organizada e automatizada. Ele garante que novos consumidores possam entender o formato das mensagens e que as mudanças nos esquemas não causem falhas inesperadas. Em resumo, o Schema Registry é uma ferramenta poderosa para manter a integridade e a evolução contínua dos dados no Kafka.

Explorando o **Kafka Streams**: Processamento de Fluxos de Dados no Kafka

Em uma aplicação baseada em Kafka, os consumidores tendem a se tornar mais complexos ao longo do tempo, muito mais do que os produtores. O que pode começar como uma simples transformação sem estado, como mascarar informações pessoais ou alterar o formato de uma mensagem para atender a requisitos internos, rapidamente evolui para operações complexas de agregação e enriquecimento.

Desafios do Processamento de Fluxos

O código do consumidor que discutimos anteriormente tem uma API bastante simples, sem muito suporte para operações complexas. Isso significa que você precisará escrever muito código adicional para lidar com:

* TimeWindows (janelas de tempo)
* Mensagens que chegam atrasadas ou fora de ordem
* Tabelas de consulta
* Agregação por chave, entre outros

Estado e Tolerância a Falhas

Operações como agregação e enriquecimento são tipicamente stateful (dependentes de estado). Esse estado é alocado na memória do heap do programa, o que é uma vulnerabilidade em termos de tolerância a falhas. Se sua aplicação de processamento de fluxos cair, o estado será perdido, a menos que você tenha um esquema para persistir esse estado em algum lugar. Criar e depurar esse tipo de solução em grande escala é extremamente complexo.

Kafka Streams

Para resolver esses problemas, o Apache Kafka oferece a Streams API. A Kafka Streams é uma API em Java que fornece acesso fácil a primitivas computacionais do processamento de fluxos, como:

* Filtragem
* Agrupamento
* Agregação
* Junção (joining)

Essa API ajuda a evitar a necessidade de escrever código de framework adicional e oferece suporte direto para grandes quantidades de estado resultantes dessas operações.

Vantagens do Kafka Streams

* Gerenciamento de Estado: Kafka Streams gerencia o estado fora do heap, persistindo no disco local e em tópicos internos do Kafka, garantindo alta disponibilidade e recuperação.
* Escalabilidade: Permite a implantação de um cluster de máquinas compartilhando a carga de trabalho de processamento de fluxos, semelhante a um grupo de consumidores.
* Integridade e Tolerância a Falhas: O estado é armazenado em disco local e em tópicos do Kafka, proporcionando recuperação rápida e segura.

Integração Simples com Microservices

Kafka Streams é uma biblioteca em Java, não uma infraestrutura dedicada separada, o que facilita sua integração em microservices. Por exemplo, um serviço de notificação de envio pode combinar eventos de envio com registros de produtos e clientes para produzir notificações de envio, enquanto expõe uma API REST para consultas síncronas.

O Kafka Streams é uma poderosa API de processamento de fluxos baseada em Java. Utiliza um paradigma funcional, onde operações complexas podem ser realizadas com chamadas simples de métodos. Isso permite criar aplicações de processamento de fluxos escaláveis e autônomas, além de enriquecer microserviços Java existentes com funcionalidades de processamento de fluxos. No entanto, o Kafka Streams é projetado para Java.

Quando o Java não é uma Opção?

E se você não utiliza Java em sua stack? Ou se deseja implantar um trabalho de processamento de fluxos puro, sem interface web ou API? Nesse caso, o ksqlDB é a solução ideal.

Introdução ao ksqlDB

Descrevemos o ksqlDB como um banco de dados de streaming de eventos para Apache Kafka. Trata-se de um banco de dados altamente especializado, otimizado para aplicações de processamento de fluxos. Ele é executado em seu próprio cluster de nós, sendo um cliente do Kafka, semelhante aos brokers do Kafka.

Funcionamento do ksqlDB

Os servidores ksqlDB são implantados em um cluster escalável e tolerante a falhas, expondo uma interface REST para as aplicações. Isso permite:

* Submissão de Jobs de Processamento de Fluxos: Esses jobs são escritos como consultas SQL, parecendo pequenos programas de processamento de fluxos.
* Consulta dos Resultados: Utilizando SQL para acessar os resultados desses jobs de processamento.

A linguagem utilizada é SQL, tornando-a acessível a desenvolvedores de diversas linguagens, já que a interface REST e a linha de comando permitem a interação com o servidor ksqlDB.

Vantagens do ksqlDB

* Independência de Linguagem: Não importa qual linguagem você use em suas aplicações, o ksqlDB suporta todas através de sua interface REST.
* Facilidade de Desenvolvimento: Pode ser facilmente executado em modo de desenvolvimento com uma imagem Docker padrão ou compilado a partir do código-fonte em uma máquina local.
* Suporte a Kafka Connect: Integração com Kafka Connect para conectar a fontes de dados externas diretamente da interface ksqlDB.

O ksqlDB é uma ferramenta poderosa que oferece uma abordagem SQL para o processamento de fluxos de dados no Kafka. Ele permite a criação de aplicações de processamento de fluxos contínuos de maneira eficiente, expondo os resultados de uma forma semelhante a um banco de dados. Ideal para casos onde o Java não é a linguagem preferida ou quando se deseja um trabalho de processamento de fluxos puro sem outras responsabilidades de serviço. Ele integra-se perfeitamente com Kafka Connect, facilitando a configuração e o gerenciamento de fluxos de dados em tempo real.