Arquitetura de Processamento de Eventos em Tempo Real:

Apache Kafka, ksqlDB e .NET

Por: Pedro de Colla Varise

Resumo

1. **Conteúdo:** Sintetizar as bases teóricas e formas práticas do Apache Kafka
2. **Tópico ou unidade de estudo:** Apache Kafka, Kafka Streams e ksqlDB
3. **Data:** 30/11/2025
4. **Objetivo:** Dominar Kafka e realizar projetos em cima da plataforma de eventos
5. **Resumo prático e códigos:** https://github.com/decolla/Masterizando-Kafka
6. **Material base:**
    a. Learn Apache Kafka - Confluent Developer
    b. GitHub - confluentinc/confluent-kafka-dotnet
    c. Apache Kafka
    d. https://training.confluent.io
    e. Livro: Mastering Kafka Streams and ksqlDB - Mitch Seymour

Definições básicas

Streaming: tecnologia que permite a transação de conteúdo digital em tempo real, sem a necessidade de baixar arquivos diretos ao dispositivo eletrônico.

Mensagem/Clientes: forma fundamental de dado (no Kafka)

ZooKeeper: Ferramenta que gerencia os clusters e os brokers, está caindo em desuso ao longo das atualizações do Kafka e do Docker.

Imagem: pacote de software executável, leve e autônomo que inclui todo o necessário para executar um aplicativo.

Broker: Servidor onde o kafka está rodando, que recebe, armazena e entrega mensagens, são de grande duração, avaliabilidade e escalabilidade, componente principal do Kafka.

Cluster: Grupo de brokers que geram uma comunicação centralizada, escalável e possibilite ser capaz de lidar com uma grande quantidade de dados.

Produtor: Responsável por gerar dados a serem armazenados ou manipulados, deve informar para qual tópico e broker quer enviar os dados, e as especificações das variáveis contidas nele.

Consumidor: Consome os dados armazenados, sendo a operação de volta do produtor, podem ler e processar dados de tópicos.

Tópicos: É onde o dado é organizado e armazenado, cada tópico pode ter vários produtores e vários consumidores, podem ser classificados como homogêneos e heterogêneos com base na forma que os dados estão armazenados.

Partição: Tópicos são divididos em partições: Logs individuais onde os dados são produzidos e consumidos, permitindo Kafka a ter um escalamento horizontal e lidar com grandes volumes de dados, permitindo um processamento paralelo. Utilizamos partições para garantir **paralelismo**. Isso permite que múltiplos consumidores (threads) processem dados do mesmo sensor simultaneamente sem gargalos.

*um tópico pode ter várias partições, mas uma partição pertence a somente um tópico

Evento: Valor-chave que armazena um dado que já ocorreu, dura exatos 7 dias no log.

Flow/Stream: Representa um corrente (normalmente de dados/informações) contínua pelos dispositivos eletrônicos modernos.

Log: Um arquivo que armazena informações sobre eventos e atividades de um sistema, aplicativo ou rede. Serve para monitoramento, diagnóstico e solução de problemas técnicos.

Replicação: É uma implementação crítica do Kafka que garante avaliação de dados e tolerância a erros

Pipelines: Série de etapas sequenciais que processam dados brutos de várias fontes até um destino final para análise.

CI/CD (Integração Contínua/Entrega Contínua): Conjunto de práticas que automatiza processos de desenvolvimento de um software

Gargalos: Ponto do sistema no qual sua performance é limitada, causando um atraso nos processos em geral.

Introdução

O Problema

Nos dias modernos, empresas e negócios geram uma vasta quantidade de dados em tempo real, como transações monetárias, atividades e serviços web, leitura de sensores,e mais. Métodos tradicionais de processamento (normalmente estáticos), normalmente apresentam problemas de eficiência e lidar com esse **"flow" contínuo de informações**, resultando em problemas de atraso, gargalos de sistema e desafios de integração.

O que é?

Imagine uma grande empresa onde departamentos necessitam de atualizações em tempo real, invés de chamadas ou e-mails, eles usam um centro de controle de mensagens para compartilhar informações de forma instantânea. **Apache Kafka** é essa mensagem que envia **dados**, ajudando a coletar, armazenar e distribuir informações em tempo real.

Apache Kafka é uma plataforma de fluxo de eventos distribuída, de código aberto, que processa dados de streaming em tempo real. Ele funciona como um sistema de mensagens de alto desempenho que permite **publicar e consumir fluxos de dados**, armazenar esses fluxos de forma durável e processá-los em tempo real, sendo amplamente utilizado para integrar sistemas, coletar dados e construir aplicações escaláveis. A plataforma serve como um centro de distribuição para conexões, onde cada produtor e consumidor conecta de forma independente a um Kafka cluster. Kafka empodera arquiteturas de modelo event-driven para ser utilizada em serviços diversos, banco de dados e sistemas analíticos.

Kafka é versátil, achando aplicações em log, coleção de métricas, análises em tempo real, e mais. No entanto, Kafka não é desenvolvido para transações ou envios complexos ou para sistemas de baixa manutenção, requer planejamento cuidadoso em particionamento, replicação, monitoramento e escalabilidade, nem sempre é necessário, mas em sua área realiza da melhor forma sua função proposta.

Modelo Event-Driven

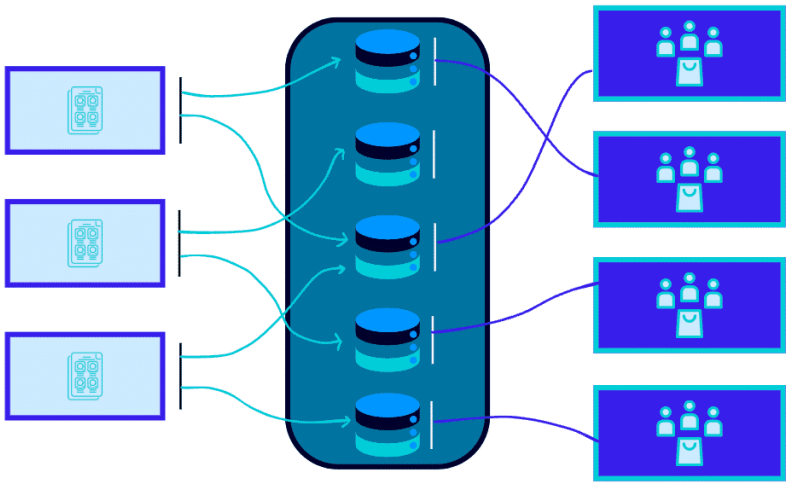
O modelo de streaming que o Kafka, invés de depender de updates ocasionais (chamadas), ele se baseia em **fluxos de eventos contínuos e ilimitados**, representando o estado atual dos dados consumidos pela aplicação, as quais rodam continuamente e são processadas à medida que chegam. Hoje em dia, aplicações modernas dependem de processamento em tempo-real para operações e rápidas decisões. Esta tecnologia consegue prover uma plataforma que conecta todo mundo a todos os eventos, um streaming em tempo real de eventos e o armazenamento de eventos para consultas futuras e confiabilidade.

“Precisamos alterar nosso pensamento onde tudo está parado, para tudo em movimento”

A Aplicação

Aplicações ("produtores") enviam fluxos de dados (eventos) para o Kafka, organizados em categorias chamadas "tópicos". Outras **aplicações ("consumidores")** se inscrevem nesses tópicos para ler e processar os dados. Essa comunicação ocorre sobre protocolo TCP e segue a arquitetura orientada a eventos. Por ser clusterizada, permite que ela funcione em um ou mais servidores.

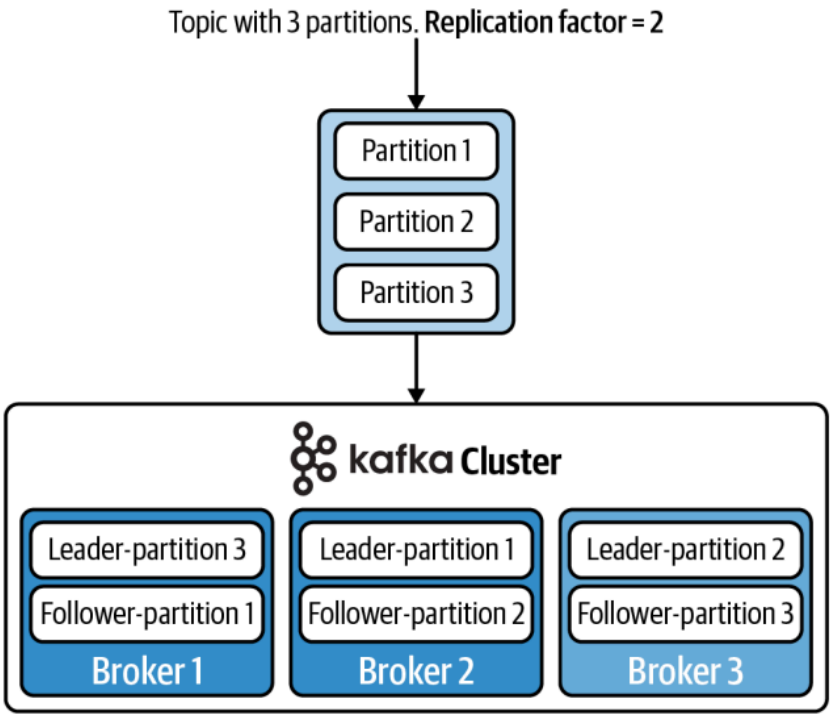
- **Decoupling:** Significa que produtores e consumidores não têm conhecimento direto ou dependência entre si, não se sabem quem está produzindo ou quem é o consumidor, e quando os dados estarão disponíveis. Esse nível de abstração permite que cada componente possa escalar de forma independente e evoluir sem prejudicar o outro



fonte: https://training.confluent.io/content

Kafka Streams e ksqlDB

Kafka Streams é uma plataforma focada em processar dados em tempo real diretamente no Apache Kafka, oferecendo controle de dados e eventos sobre o fluxo. Já o ksqlDB é uma plataforma baseada em SQL sobre o Kafka Streams que permite manipular e visualizar dados de fluxo de forma mais simples, usando uma linguagem familiar ao de banco dados relacionais. Veremos mais sobre essas ferramentas nos tópicos adiantes.



fonte: Mastering Kafka Streams and ksqlDB - Mitch Seymour

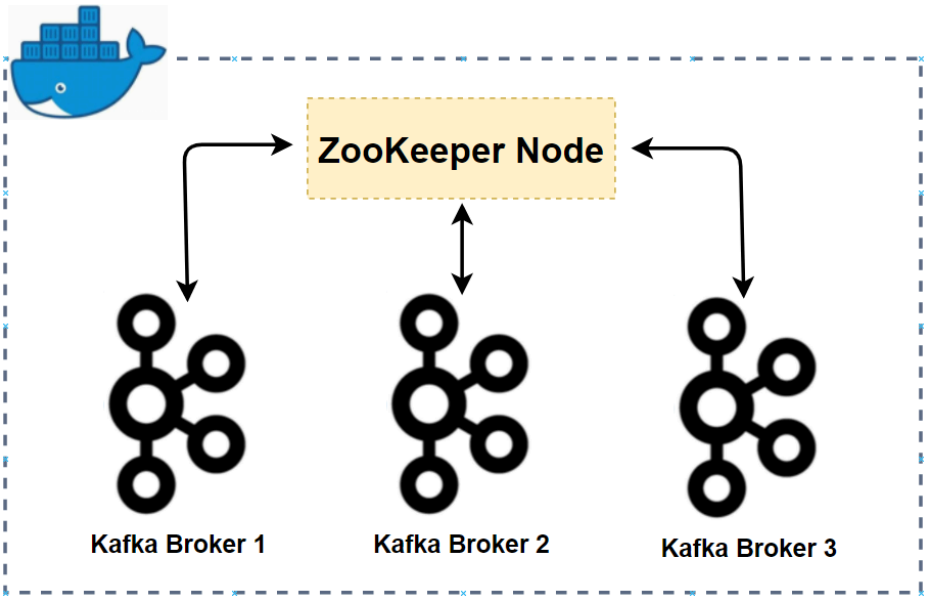
Conceito de uso

Tratar de dados em fluxo é um conceito aplicável para diversas situações como: monitoramento, filtragem, modificação, manipulação e otimização de dados. Kafka permite isso sem o uso de um banco estático de dados e consultas contínuas a ele, o usuário pode visualizar um dado no momento que ele nasce, e posteriormente decidir o que fazer com tal.

Infraestrutura de Containers

Container para o Kafka

Para a execução do Kafka e do ksqlDB, usaremos o **Docker,** uma plataforma de código aberto que permite aos desenvolvedores construir, empacotar e executar aplicações em **contêineres** isolados. Esses contêineres são unidades padronizadas que incluem o código da aplicação com todas as suas dependências, bibliotecas e configurações, garantindo que o aplicativo funcione de forma consistente em diferentes ambientes. Utilizaremos essa função para o Kafka em uma máquina ou servidor local, a imagem abaixo representa uma representação visual do processo.



fonte: https://jaehyeon.me/blog/2023-05-04-kafka-development-with-docker-part-1/

Networking

O funcionamento do networking no Docker permite que aplicações de fora do ambiente Docker acessem serviços expostos e que os containers dentro da mesma rede se comuniquem usando nomes de serviço, algo fundamental na comunicação de microsserviços. Dentro de uma rede definida pelo usuário, os containers podem se comunicar entre si usando seus nomes de serviço ou nomes de container, graças ao serviço de DNS interno do Docker.

Utilizando o Kafka

Indo diretamente para a forma prática, usaremos o Kafka por meio do **Docker,** pois permite realizar testes e rodar aplicações de forma rápida, caso não tenha instalado siga por este link. Após instalado, copie o código no git abaixo e salve como um arquivo na pasta desejada para o projeto como "**docker-compose.yml**":

(O código abaixo pode apresentar problemas ou divergências devido a diferença de versões ou configurações do terminal, nos links das referências há outros docker compose para serem usados)

https://github.com/decolla/Masterizando-Kafka/blob/main/docker-compose.yml

Após copiado e com o Docker instalado e executando, execute os comandos no **terminal** de seu sistema operacional aberto no caminho da pasta (> cd caminho/pasta/):

> docker compose up -d

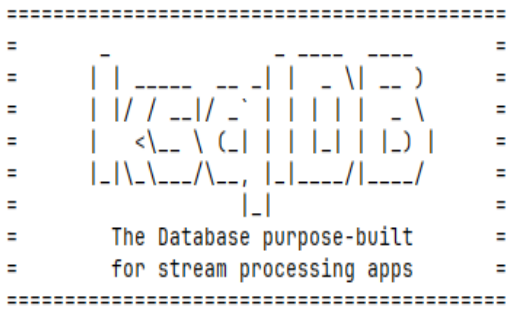
O resultado deve ser:

[+] Running 4/4
✔ Container zookeeper Running
✔ Container broker Running
✔ Container ksqldb-server Running
✔ Container ksqldb-cli Running

Com isso o docker compose funcionou! Agora execute o ksqlDB:

> docker exec -it ksqldb-cli ksql http://ksqldb-server:8088

O resultado deve ser:



Copyright 2017-2022 Confluent Inc.

CLI v0.28.2, Server v0.28.2 located at http://ksqldb-server:8088
Server Status: RUNNING

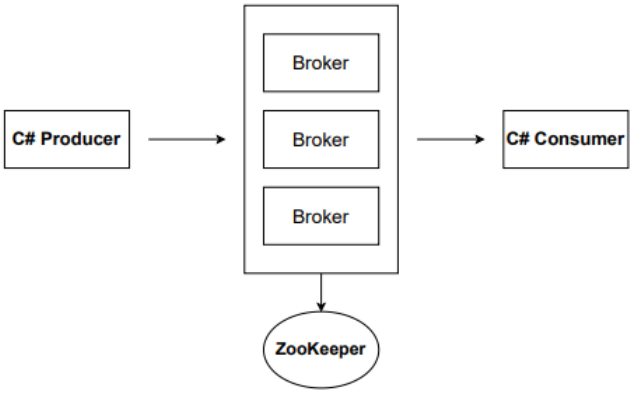
Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>

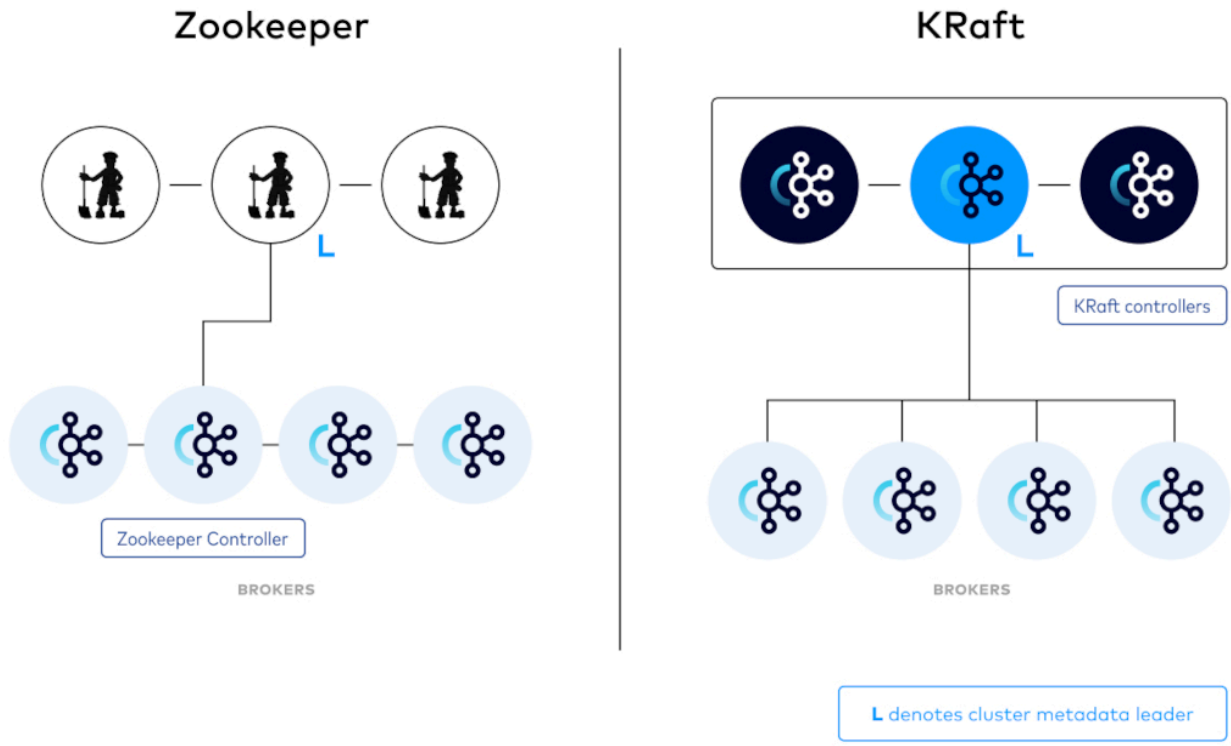
Com isso o ksqlDB deve estar **rodando perfeitamente no servidor especificado!**

Funcionamento e ZooKeeper

Usar o ZooKeeper (uma das instâncias do docker compose) junto ao Docker, envolve executar uma imagem para objetivos de desenvolvimento e teste. Essa função junta ao Apache Kafka permite orquestrar dois serviços distintos, de forma que existam comunicadores para ambiente externo (usuário) e interno (inter-broker).



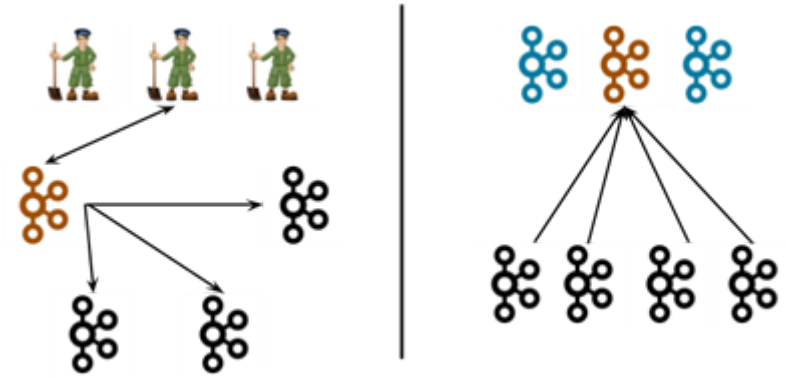
No entanto, versões modernas do Kafka conseguem operar em KRaft mode, no qual elimina a dependência do ZooKeeper para o controle de metadata. Isso pode simplificar o desenvolvimento removendo a necessidade de um espaço separado para um cluster do ZooKeeper, se tornando uma ferramenta mais rápida, simplificada e com melhor propagação e escalabilidade.



fonte: https://www.donaldsebleung.com/blog/20250119-operating-kafka-at-scale-with-strimzi

Talvez se pergunte por que se está usando o Zookeeper neste caso, e a explicação é a seguinte: o ZooKeeper é originalmente o serviço de controle de metadata usado para o Kafka, devido isso, muitas aplicações e sistemas usam o versões do Kafka ou programas ligados a ele que dependem do ZooKeeper para funcionar, devido o KRaft ser uma implementação moderna do Kafka.

Além disso, ZooKeeper é uma forma estável e utilizada em ambientes de coordenação de sistemas por um longo período de tempo, usado por diversas aplicações além do Kafka. No entanto, é extremamente recomendável o uso do KRaft para o Kafka pois simplifica o desenvolvimento e configuração do ambiente, eliminando a necessidade de dois sistemas separados com diferentes configurações. Invés disso, o próprio Kafka lida com a metadata, reduzindo erros e complexidade operacional, operando de forma similar ao jeito de funcionamento dos produtores e consumidores.



fonte:
https://medium.com/@husain.ammar/breaking-free-from-zookeeper-why-kafkas-kraft-mode-matters-90819c3c3e57

Interface de Programação de Aplicativos (API)

Observações importantes

Para a codificação e implementação do Kafka e um programa vinculado a ele, estarei usando especificamente a linguagem de programação **C#** com a importação da biblioteca externa `Confluent.Kafka`, que pode ser usada após executar o seguinte comando no terminal do projeto:

```
> dotnet add package Confluent.Kafka
```

* caso não tenha criado um projeto ainda, execute no terminal da pasta desejada:

```
> dotnet new console
```

Além desta biblioteca, estarei usando outra biblioteca externa, de preferência pessoal, para facilitar o tratamento de dados pelo formato **JSON**, que deixa eles de forma mais legível e funciona de forma agradável junto ao Kafka. A biblioteca `Newtonsoft.Json` pode ser obtida após também executar o seguinte comando:

```
> dotnet add package Newtonsoft.json
```

Producer e Consumer API

A chave para a manipulação de dados no kafka é o produtor, o qual é usado para publicar mensagens para um tópico no Kafka. O consumidor escuta os dados em tempo real no Kafka, sendo independente do produtor. A estrutura dos dois são bem semelhantes e possuem parâmetros como:

- **Nome do tópico** para publicar mensagens
- **Chaves e Valores** serializados ou não serializados
- As **configurações chaves**
- Configurações de **segurança**

*Os códigos podem ser vistos e instalados no link:
https://github.com/decolla/Masterizando-Kafka/tree/main/Programa-base

Configurações: são usadas para setar configurações do produtor, como: Id, bootstrap server e os serializadores, que convertem estruturas complexas de dados em forma legível. Serão passadas para o produtor.

```
// configurações simples de um produtor

var config = new ProducerConfig{ BootstrapServers = "localhost:9092"
};
```

```
// configurações simples do consumidor

var config = new ConsumerConfig

{

   BootstrapServers = "localhost:9092", // servidor de acesso

   GroupId = "integration-consumer-group", // identidade do grupo

   AutoOffsetReset = AutoOffsetReset.Earliest // ler desde o começo se
não souber de onde parou

};
```

Produtor e consumidor: Cria o produtor ou consumidor e passa as configurações e suas propriedades, nele que serão usados as funções de manipulação do Kafka.

```
// using -> loop infinito que garante que a conexão feche se der erro

// construindo um produtor com ProducerBuilder
```

```
using (var producer = new ProducerBuilder<Null,
string>(config).Build())
```

```
using (var consumer = new ConsumerBuilder<Ignore,
string>(config).Build())
```

Produzir/Consumir: Aqui está o núcleo do ambiente de produção e consumo, onde dados deverão ser populados ou consumidos, toda a manipulação de dados gira em torno dessas operações fundamentais do Kafka. Lembre-se que estarei usando o formato de JSON para os dados armazenados.

A parte de segurança está implementada juntamente às operações, usando try/catch para tratamento de erros.

```
// OPERAÇÃO DE PRODUZIR
while (true)
   {
       // inicializar um novo dado
       var dadosSensor = new DadosSensor
       {
           SensorId = "sensor-01",
           NivelAgua = random.Next(1000, 9999),
           Botao = false,
           TimeStamp = DateTime.UtcNow // data e hora atual
       };

       var dadosCorrente = new DadosCorrente
       {
           // usar o mesmo id para o join funcionar
           SensorId = "sensor-01",
           ValorCorrente = random.Next(0, 50),
       };

       // transformar em JSON
```

```
        string valorJsonSensor = JsonConvert.SerializeObject
(dadosSensor);

        string valorJsonCorrente = JsonConvert.SerializeObject
(dadosCorrente);



        // kafka espera um objeto do tipo Message

        var mensagemSensor = new Message<Null, string> { Value =
valorJsonSensor };

        var mensagemCorrente = new Message<Null, string> { Value =
valorJsonCorrente };



        // adiciona o dado no kafka Stream

        // o await é crucial para ser assíncrono

        await producer.ProduceAsync(nomeTopicoSensor, mensagemSensor);

        await producer.ProduceAsync(nomeTopicoCorrente,
mensagemCorrente);



        //Console.WriteLine($"Enviado Sensor: {valorJsonSensor}");

        //Console.WriteLine($"Enviado Corrente: {valorJsonCorrente}");



        // espera um pouco para continuar

        await Task.Delay(1000);

    }

}
```

*Na operação de produção acima, é possível perceber que convertemos os dados iniciados para o formato JSON, mesmo que o Kafka aceite bytes puros (array) como forma de dado, escolhi usar o padrão JSON para manter a interoperabilidade com o ksqlDB, que nativamente faz parsing com JSON.

```
// OPERÇÃO DE CONSUMO

    using (var consumer = new ConsumerBuilder<Ignore,
string>(config).Build())

    {

        // subscribe nos dois tópicos ao mesmo tempo usando uma Lista
```

```
        consumer.Subscribe(new List<string> { nomeTopicoSensor,
nomeTopicoCorrente, nomeTopicoAlerta });


        while (true)
        {
            try
            {
                var resultado = consumer.Consume();


                // verifica de qual tópico veio
                if (resultado.Topic == nomeTopicoSensor)
                {
                    //Console.WriteLine($"[Água] Recebido:
{resultado.Message.Value}");
                }
                else if (resultado.Topic == nomeTopicoCorrente)
                {
                    //Console.WriteLine($"[Corrente] Recebido:
{resultado.Message.Value}");
                }else if (resultado.Topic == nomeTopicoAlerta)
                {
                    //Console.WriteLine($"[!!! PERIGO !!!] O ksqlDB
detectou: {resultado.Message.Value}");
                }
            }
            catch (Exception e)
            {
                Console.WriteLine($"Erro ao ler: {e.Message}");
            }
        }
    }
```

Engenharia do código

Esta arquitetura da aplicação C# foi desenhada para ser **Full-Duplex**, o que permite a troca de dados de forma simultânea e bidirecional em um único canal, onde cada parte (consumidor e produtor) podem enviar e receber informações ao mesmo tempo. Tal característica é possível graças ao padrão Non-Blocking I/O, um método de operação de entrada/saída que permite que uma thread possa executar suas tarefas enquanto outras operações também estão ocorrendo no sistema. Esse padrão também é usado no while( ).

Essa abordagem assíncrona garante que picos de dados vindos dos sensores não causem latência na recepção de alertas críticos , mantendo a aplicação responsiva mesmo sob carga.

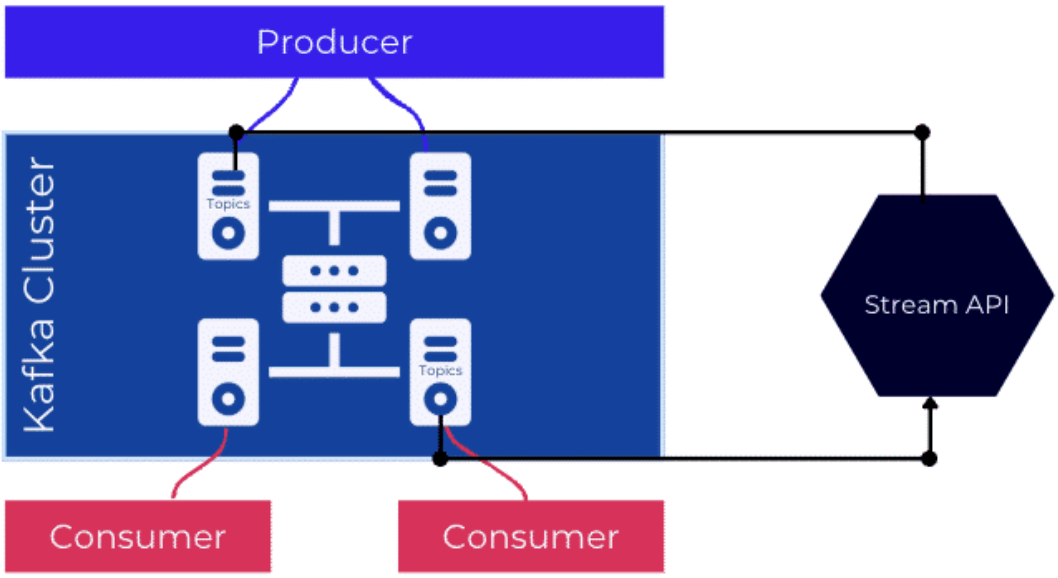
```
    public static async Task Main(string[] args)
    {
        Console.WriteLine("Iniciando Aplicação...");


        // Executa as funções em paralelo
        var tarefaProdutor = Task.Run(() => RodarProdutor());
        var tarefaConsumidor = Task.Run(() => RodarConsumidor());


        // a Main espera quando as duas terminarem (o que nunca vai
acontecer)
        await Task.WhenAll(tarefaProdutor, tarefaConsumidor,
tarefaControlador);
    }
```

Como podemos ver no código acima, há a chamada de **dois processos (consumir e produzir) ao mesmo tempo**, de forma que a ingestão de dados (producer) não bloqueie a leitura de alertas (consumer), permitindo alta vazão (throughput) na mesma instância de serviço apresentada no programa.

Apache Kafka Streams

Kafka Streams é uma poderosa biblioteca usada para construir aplicações de **processamento de streams em tempo real**, permite escalabilidade e tolerância a erros para o

processamento de grandes fluxos de dados, tudo isso em uma estrutura simples sem necessitar de clusters adicionais.



fonte: https://training.confluent.io/content

Kafka Connect

Apache Kafka Connect é uma ferramenta e um framework para integração de dados para escalabilidade e apoio para conexões externas, simplificando a integração streaming de dados que monitora e manipulando os objetos armazenados. A ferramenta balanceia automaticamente os conectores e as tarefas pelos processadores, armazenando dados em tópicos do Kafka, e mantendo o código leve, sendo compatível com qualquer ferramenta de orquestração externa, facilitando o uso do Kafka. Seu funcionamento é baseado em **rodar conectores no hardware de forma independente** dos brokers, garantindo escalabilidade e tolerância a erros.

Processamento de Fluxo em Tempo Real

Para esta seção estarei usando diretamente as ferramentas do ksqlDB, com os códigos/cláusulas SQL executados no terminal do ksql juntamente com o fluxo de dados gerados pelo programa destacado no capítulo da API.

Os códigos a seguir foram realizados para satisfazer as condições do programa apresentado anteriormente, com interação aos tópicos declarados nas funções. A criação dos tipos pode variar de acordo com as exigências da aplicação

Criar uma Stream:

```
ksql> CREATE STREAM monitoramento_agua (
>          SensorId VARCHAR,
>           NivelAgua INT,
>          Botao BOOLEAN,
>          TimeStamp VARCHAR
>) WITH (
>           KAFKA_TOPIC = 'INTEGRA_DADOS_NIVEL_AGUA_GUARAPIRANGA_PT1',
>          VALUE_FORMAT = 'JSON'
>);
```

Aqui é criada uma stream que armazena uma string(VARCHAR), um número inteiro (INT), um boolean e um valor temporal que é armazenado no JSON como uma forma de string. Passamos também o tópico pelo qual os dados estão sendo enviados e a forma de leitura dos valores, que neste caso é JSON.

Consultar o fluxo de dados de uma Stream:

```
ksql> SELECT * FROM monitoramento_agua EMIT CHANGES;
```

Nesta linha pegamos todos os valores do tópico de uma dada stream e visualizamos o fluxo de dados ocorrendo

Mostrar as Streams criadas:

```
ksql> SHOW STREAMS;
```

Com esse código, podemos verificar quais streams foram criadas no sistema.

Deletar uma Stream:

```
ksql> STREAM nome_stream;
```

Com esse código, podemos deletar uma stream no sistema.

Aplicar filtragem de dados:

```
ksql> SELECT NivelAgua, Botao
> FROM monitoramento_agua
> WHERE NivelAgua > 8500 OR ( Botao = true AND NivelAgua < 2000)
> EMIT CHANGES;
```

Neste código verificamos o nível da água e o botão da stream monitoramento_agua, e filtramos os resultados dessas duas variáveis baseado na condição: o nível da água ser maior que 8500 ou menor que 2000 se o botão estiver ativo. Tal aplicação vai resultar na tabela com somente os dados que passam na condição

Visualizar Janelas Temporais:

```
ksql> SELECT SensorId, FUNCAO ( NivelAgua ) AS FuncaoAgua
> FROM monitoramento_agua
> WINDOW TIPO_JANELA ( SIZE X MEDIDA )
> GROUP BY SensorId
> EMIT CHANGES;
```

FUNCAO: Qualquer fator de agregação, os abaixo são os mais utilizados.

- COUNT(*) = contagem de vezes de uma coluna
- MAX (coluna) = máximo dos valores
- MIN (coluna) = mínimo dos valores
- AVG (coluna) = média dos valores
- SUM (coluna) = soma dos valores

TIPO_JANELA: Como a janela se comporta ao decorrer do tempo especificado, que representa o intervalo de atualização de dados. Isso impede que ocorra certos problemas na análise de dados como a média infinita, onde uma janela de 30 segundos resolve isso.

- TUMBLING ( ): tem tamanho fixo, funções são contínuas e não se sobrepõe.
- HOPPING (, ADVANCE BY X MEDIDA): tem tamanho fixo, funções são contínuas e se sobrepõe.

X MEDIDA: representa um valor (**X** = 10, 20, 30) e a medida de tempo (**MEDIDA** = SECONDS, MINUTES, HOURS) que a janela sofrerá uma atualização, portanto, os valores passados para a função serão somente os que forem lidos na faixa de tempo especificada.

Joins (Inner Join):

```
ksql> SELECT
>          S1.SensorId,
>          S1.NivelAgua,
>          S2.ValorCorrente
> FROM monitoramento_agua S1
> INNER JOIN monitoramento_corrente S2
> WITHIN ( X SECONDS, X SECONDS )
> ON S1.SensorId = S2.SensorId
> WHERE S1.NivelAgua > 8500 AND S2.ValorCorrente > 30
> EMIT CHANGES;
```

Neste Join, selecionamos primeiramente as variáveis que serão consumidas e de qual stream elas se originam, sendo SensorId e NivelAgua da stream monitoramento_agua e ValorCorrente da monitoramento_corrente (uma stream criada da mesma forma somente com dados diferentes).

Depois de declarar quais são as variáveis e as streams consumidas no join, declaramos o intervalo de tempo para procurar um elemento pareável entre as streams, sendo o primeiro valor antes da vírgula o tempo analisado antes e o outros segundos futuros. O elemento pareável é declarado logo em seguida, sendo o SensorId das streams, o que exige que tenham casos onde os valores são idênticos para ocorrer o join.

Além de tudo isso, há uma filtragem no join no qual o nível da água do monitoramento da água tem que ser maior que 8500, e o nível da corrente no monitoramento da corrente maior que 30.

Automação e controle via API

Como vimos antes, para realizar as operações no Kafka e no ksqlDB, precisamos digitar comandos SQL no terminal embutido. No entanto, é possível realizar a automação desses processos (filtragem, joins, janelas temporárias) com o uso de um **Controlador HTTP**, que basicamente usará do HttpClient do C# juntamente ao servidor do Kafka, para comandar a infraestrutura dos streams de forma dinâmica, sem intervenção humana direta.

O código abaixo representa um “Control Plane”, que realizará uma filtragem onde a água seja maior que 9000, de modo que seja implementado automaticamente quando a ação é liderada pelo usuário.

*na Main:

```
var tarefaControlador = Task.Run(() => RodarControlador());

public static async Task RodarControlador()
{
   // cliente http para falar com o ksqlDB
   using var client = new HttpClient();
   // define
   client.BaseAddress = new Uri("http://localhost:8088");

   while (true)
   {
       Console.WriteLine("1 - Criar Filtro | 2 - Apagar Filtro");
       var opcao = Console.ReadLine();

       string sqlCommand = "";
       if (opcao == "1")
       {
           sqlCommand =
               "CREATE STREAM ALERTA_PERIGO " +
               "AS SELECT * " +
               "FROM monitoramento_agua " +
               "WHERE NivelAgua > 9000;";
       }
       else if (opcao == "2")
       {
           sqlCommand = "DROP STREAM ALERTA_PERIGO DELETE TOPIC;";
       }
```

```
        // se o input foi valido
        if (string.IsNullOrEmpty(sqlCommand)) continue;

        // monta o JSON
        string jsonCommand = JsonConvert.SerializeObject(new
        {
            ksql = sqlCommand,
            streamsProperties = new {}
        });

        // cria o conteúdo http
        var content = new StringContent(jsonCommand,
System.Text.Encoding.UTF8, "application/json");

        // mandar
        try
        {
            var response = await client.PostAsync("/ksql", content);

            // repsosta de sucesso ou erro
            string respostaTexto = await
response.Content.ReadAsStringAsync();

            Console.WriteLine($"Status: {response.StatusCode}");
            Console.WriteLine($"Resposta: {respostaTexto}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Erro: {ex.Message}");
        }
    }
```

```
}
```

Esta implementação permite realizar uma criação de uma stream com filtragem digitando apenas o número da operação que você quer, se é aplicar o filtro ou deletá-lo, representado na seguinte interface:

1 - Criar Filtro | 2 - Apagar Filtro

A robustez do controlador depende da formatação estrita do JSON, onde a captura de exceções HTTP nos permite diagnosticar falhas de sintaxe SQL sem derrubar a aplicação em qualquer caso de erro ou mal funcionamento.

Administração

A administração de clusters no Kafka é algo de extrema importância que garante disponibilidade e otimização da performance do sistema como um todo, por meio de:

- Manutenção de clusters
- Manipulação das configurações dos Brokers
- Monitoramento e lidar com problemas
- Controle de segurança
- Replicação de dados e backup
- Aprimoramento de performance e otimização
- Melhorias e atualizações
- Colaboração e suporte

Clusters dinâmicos e estáticos

Clusters podem ser configurados de duas formas para alcançar a melhor performance possível, sendo elas:

- **Estática**: Essa configuração exige que ela seja a mesma entre todos os brokers do cluster, de forma padronizada, é necessário reiniciar o broker toda vez que forem feitas alterações nas configurações das propriedades.
- **Dinâmica:** Tal tipo de configuração permite que desenvolvedores e administradores criem diferentes configurações para diferentes brokers ou tópicos, essas configurações sobrescrevem as estáticas quando criadas.

Performance

A arquitetura do Kafka é responsável pela forma que será desenvolvido o design, implementação e o controle do streaming de eventos em larga escala, garantido que as pipelines funcionem de forma esperada, essas modificações exigem:

- Design de sistema e arquitetura
- Controle do flow de dados
- Segurança
- Recuperação de desastres e quedas
- Desenvolvimento de integração CI/CD
- Melhorias contínuas

Throughput e Latência

- **Throughput:** Se refere à quantidade de gravações mandadas de um produtor para um Kafka cluster em um período de tempo, normalmente medido de gravações (records) por segundo ou bytes por segundo.
- **Latência:** É o tempo que se leva para uma mensagem viajar de um produtor até um cluster, latências baixas são casos críticos para aplicações de evento em tempo real, mas conseguem vir em grande parte dos casos em um throughput extenso.

Testando Performance do Produtor e do Consumidor

Criar instâncias de teste dos seus produtores e consumidores atuais, de forma controlada, podendo evitar gargalos do cluster. O throughput do consumidor depende de vários fatores, como:

- Tempo de processamento por mensagem
- Números de mensagens recebidas por execução
- Tamanho das mensagens

Uma das principais formas de garantir escalabilidade e performance do seu cluster é adicionando mais brokers e distribuídos partições entre eles, melhorando a taxa e trabalho sem comprometer a performance do sistema

Conclusão

Neste projeto, exploramos os fundamentos do Apache Kafka e suas ferramentas, saímos de uma arquitetura teórica para uma implementação funcional capaz de ingerir, filtrar e alertar sobre dados hídricos em tempo real. Demonstramos que o uso de ksqlDB reduz a complexidade de código comparado a soluções tradicionais, enquanto o ecossistema .NET provê a performance necessária para a camada de aplicação. Apache Kafka é certamente uma plataforma de extremo poder no contexto da tecnologia moderna, onde enormes quantidades de dados circulam por toda internet em tempo real, e o Kafka permite manipular isso.