Apache Kafka para Iniciantes

Do primeiro tópico às Transações

Autor: Fábio José de Moraes
Revisão: Dagmar F. Napolitano

18 September 2020

Conteúdo

Preparação

Antes de iniciar este módulo é importante garantir que seu ambiente: pc, notebook, workstation, esteja preparado para execução dos nossos laboratórios. Então, com base nas orientações abaixo, realize as seguintes preparações.

- **Máquina com configuração mínima de 4GB RAM e 50GB de espaço no HD**
- **Conexão banda-larga com a internet**
- **Java instalado nas versões 1.8 ou 11**
- **Usuário no sistema operacional com permissão para executar serviços de rede**
- **Usuário no sistema operacional com permissão para realizar download de arquivos .zip**

Verificar versão Java

Windows:

- Abrir o prompt de comando
- Digitar este comando e teclar enter: `java -version`
- A versão do Java estará correta se o resultado for similar ao descrito abaixo:

```
openjdk version "1.8.0_242"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_242-b08)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.242-b08, mixed mode)
```

Linux:

- Abrir o terminal
- Digitar este comando e teclar enter: `java -version`
- A versão do Java estará correta se o resultado for similar ao descrito abaixo:

```
openjdk version "11.0.6"
OpenJDK Runtime Environment (build 11.0.6+10-post-Debian-1bpo91)
OpenJDK 64-Bit Server VM (build 11.0.6+10-post-Debian-1bpo91)
```

Instalar Java 1.8 ou Java 11

Java **1.8** para Linux.

Debian, Ubuntu, etc:

```
sudo apt-get install openjdk-8-jdk
```

Fedora, Oracle Linux, Red Hat Enterprise Linux, etc.

```
su -c "yum install java-1.8.0-openjdk-devel"
```

Java **1.8** para Windows.

- 32bits: http://bit.ly/java_1_8-x32
- 64bits: http://bit.ly/java_1_8-x64

Java **11** para Linux.

> procure no google: *how to install openjdk 11 DISTRO NAME install*

Java **11** para Windows.

- 64bits: http://bit.ly/java_11-x64

Adicionais

Se você é usuário do windows, recomendamos a instalação dos seguintes aplicativos:

- 7-Zip: https://www.7-zip.org/download.html
- Notepad++: https://notepad-plus-plus.org/downloads/v7.8.5/

Para Labs, recomendamos algum editor para você visualizar o código-fonte:

- VS Code: https://code.visualstudio.com/download
- Atom: https://atom.io/
- Notepad++: https://notepad-plus-plus.org/downloads/v7.8.5/

Conceitos Gerais

A conceituação geral é importante para que todas as pessoas tenham em mente diversos detalhes ligados à computação distribuída, que são utilizados com frequência quando falamos sobre Kafka.

Vínculos de Transporte

O *Transport Binding* é também conhecido como *Protocol Binding*, mas aqui o chamaremos de Vínculos de Transporte, estão relacionados ao meio em que os dados trafegam e definem se esses dados são pacotes binários ou texto puro e se trafegam em modo seguro.

Porém, o importante é sempre ter em mente que o meio de transporte não presume qualquer semântica de comunicação, formato de dados, padrão de desenvolvimento ou integração.

E na computação distribuída os vínculos de transporte são importantíssimos, pois toda comunicação acontece via rede e sem eles não haveria abstração de formatos, dentre outros aspectos.

Exemplificando:

> *Como seria se HTTP transportasse apenas html?*

- HTTP
- GRPC
- RSOCKET
- AMQP (Advanced Message Queuing Protocol)
- MQTT (Message Queuing Telemetry Transport)
- KAFKA

Formato de Dados

Data Format, que aqui chamaremos de **Formato de Dados**, trata de qual formato será utilizado para levar dados de um ponto "A" para um ponto "B" através de um Vínculo de Transporte. Ele define características como sistema de tipos, esquema externo ou embarcado, estrutura dos registros, entre outros detalhes.

Mas o importante é compreender que o formato de dados não determina ou limita a semântica de comunicação.

Exemplos:

- JSON
- AVRO
- PROTOCOL BUFFERS
- APACHE THRIFT

- CBOR
- XML

Figura 1: Vínculo de Transporte + Formato de Dados

Mensagens & Eventos

Hoje temos uma infinidade de termos, definições, padrões e *buzzwords*. Isso gera mal-entendidos sobre o que é **mensagem** e o que é **evento**. Saber quando estamos produzindo e/ou consumindo mensagens ou eventos é crucial para solucionar corretamente os problemas de negócio através da computação distribuída.

Bem, mas agora que vimos vínculos de transporte e formatos de dados, vamos estabelecer a Semântica da Comunicação. E aí que entram as mensagens e os eventos.

Mensagens são . . .

As mensagens definem uma semântica na comunicação onde há **intenção** nos dados produzidos, ou seja, existe a **expectativa** de que eles passem por algum tipo de processamento, produzindo um resultado em algum momento no futuro. Essas mensagens acontecem através de qualquer formato de dados, sobre qualquer vínculo de transporte.

> Mensagens são dados carregando a intenção de que algo aconteça no futuro.

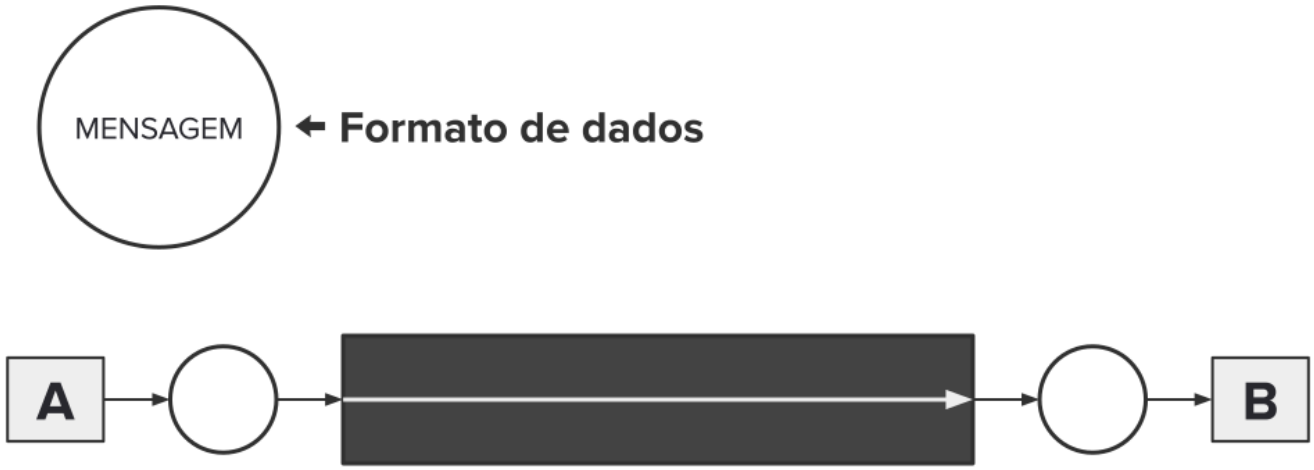

Figura 2: Mensagens com formato de dados e vínculo de transporte.

Ao produzir (A) (Figura 2) uma mensagem, utilizando um formato de dados e um meio de transporte, existirá a intenção, a expectativa que ela seja processada (B) e tenha algum resultado. Pensando em uma linha do tempo, a mensagem remete a algo que deverá acontecer no futuro.

O efeito dessa mensagem poderá ser uma resposta, continuação de um fluxo, execução de trabalho (*job*), uma consulta, uma atualização, etc. Mas sempre existirá a intenção embutida nos dados produzidos, ou seja, essa é a **natureza semântica** das **Mensagens**.

Eventos são . . .

Os eventos definem uma semântica na comunicação onde os dados produzidos transmitem um **fato** ocorrido, ou seja, em nenhum momento existe a expectativa que ele seja processado ou exista algum resultado. Esses eventos acontecem através de qualquer formato de dados, sobre qualquer vínculo de transporte.

> Eventos são dados relatando um fato ocorrido em algum ponto no passado.

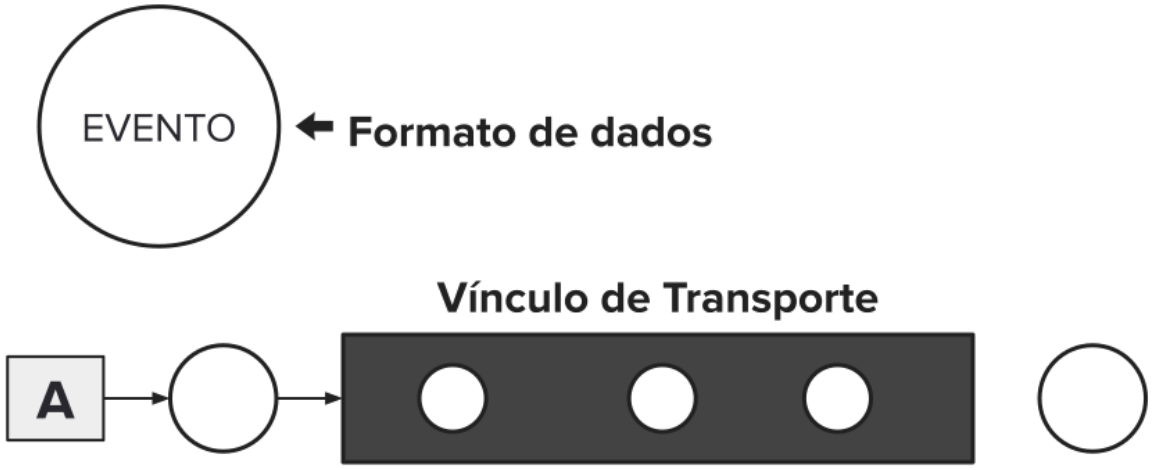

Figura 3: Ilustrando evento com Formato de dados e Vínculo de transporte

Um ponto "A" (Figura 3) ao produzir **eventos**, o envia utilizando um formato de dados e um vínculo de transporte e ao fazer isso, "A" só está transmitindo o **fato** ocorrido e não há preocupação se haverá processamento destes dados mesmo que cheguem a um broker persistente.

(Exemplificando)...

Figura 4: Avro com Kafka para eventos

Será muito comum utilizar avro como formato de dados e Kafka como vínculo de transporte, ambos em uma semântica de comunicação para eventos e para mensagens também.

Idempotência

> Operações que podem ser aplicadas sem que o valor do resultado se altere após a aplicação inicial.

Wikipedia

Qualquer operação que seja executada uma ou cem vezes, manterá o resultado final mesmo após a aplicação inicial. Como exemplo, usaremos uma operação de débito no valor de `R$ 100,00` identificada por `d598`, se esta operação for idempotente, mesmo que seja executada dez vezes, somente `R$ 100,00` serão debitados e não `R$ 1.000,00`.

Note que a idempotência é construída em torno de algo que possa ser validado, neste caso um **identificador**. No exemplo acima, se todas as dez operações de débito possuíssem identificadores diferentes não seria possível computar a idempotência, pois, semanticamente seriam coisas completamente diferentes.

Padrões de Entrega

Os padrões de entrega estão presentes tanto na produção, quanto no consumo de dados. Na produção, estes padrões garantem que o Kafka receberá os dados e tratará de processá-los para persistência. Já no consumo, eles garantem que os dados chegarão às instâncias que os requisitaram.

1 - At-least-once - Pelo-menos-uma-vez

Neste padrão "Pelo menos uma vez" os dados consumidos, em virtude de alguma situação, quando processados poderão ser consumidos novamente. No Kafka isso pode acontecer caso o *offset* consumido e processado não seja efetivado, ou seja, não aconteça o *commit*.

Exemplificando o "pelo menos uma vez"

- foram consumidos os dados correspondentes ao *offset* de `1` a `10`;
- eles foram processados;
- mas antes de ser efetivado o serviço falhou e derrubou o consumidor;
- quando o serviço voltou, ele se conectou ao Kafka para consumir dados;
- novamente, os dados de `1` a `10` foram enviados porque o Kafka não recebeu sua efetivação;
- porém, eles já foram processados e não haverá dupla-execução;
- portanto, o serviço deverá implementar **idempotência**
- só com **idempotência** é possível operar com o padrão de entrega *At-least-once*.

2 - Exactly-once - Exatamente-uma-vez

Neste padrão "Exatamente uma vez" os dados são consumidos, e processados apenas uma vez, ou seja, existirá uma garantia que os dados que já foram processados foram efetivados. Com Kafka podemos fazer isso através da **Transaction API**, que veremos em detalhes neste módulo.

Apache Kafka 101

Kafka é uma plataforma distribuída para *streaming*, como definido pelo seu site oficial, que escala horizontalmente e provê alta disponibilidade.

Streaming

> São dados produzidos continuamente por diferentes fontes.

Streaming of everything: a próxima onda de inovação, onde já se fala até de streaming de dinheiro.

A característica principal de um *stream* é a impossibilidade de determinar seu início ou seu fim, ou seja, quando estamos trabalhando com Kafka Streams por exemplo, trabalhamos com janelas de tempo, *time window* em inglês. Isso permite o processamento contínuo dos registros, visto que podem haver milhões deles passando pelos tópicos. E isso torna inviável qualquer possibilidade de leitura desde o início até o *fim*, pois este *fim* é não-determinístico, ou seja, não é possível determinar se o fluxo de registros terminou.

Será bastante comum falarmos sobre *streaming* quando trabalharmos com Kafka.

Kafka é . . .

- **Plataforma tolerante à falhas**: sua arquitetura e operação foi idealizada para ser tolerante às falhas na rede e na replicação de dados, bem como para a não perda de dados.
- **Para alta vazão de dados (taxa de transferência)**: a forma como os dados são gravados no sistema de arquivos é desenhado para alta velocidade na escrita e na leitura, por isso não existem buscas ou filtros para consulta de dados.
- **O melhor dos dois mundos** - *tópicos* e *filas*: o melhor dos tópicos, porque vários destinos podem consumir os dados presentes no kafka. E o melhor das filas, porque ele garante que dado um registro, jamais este será enviado para dois ou mais consumidores diferentes em um mesmo grupo de consumo. Isso quer dizer que, àquele registro será enviado apenas a um destino no grupo, assim como acontece com consumidores em ferramentas tradicionais para filas.
- **Garante a entrega de dados pelo-menos-uma-vez**: esse é o padrão de entrega *out-of-the-box* para consumidores no kafka. Há uma garantia de que os dados sempre chegarão, isso uma ou várias vezes.
- **Confiável, armazenamento persistente e durável**: quando produzidos nos kafka, os registros estarão lá por 7 dias na configuração padrão, ou *ad aeternum* se assim for desejado.
- **Código fonte aberto**: Kafka é uma ferramenta de código fonte aberto muito popular na comunidade, com cerca de 15 mil estrelas no Github. E inúmeras corporações, como Confluent, Red Hat, Microsoft e Google contribuem para sua evolução, além de milhares de outras que são suas usuárias.

Kafka não é . . .

- **Uma ferramenta que se preocupa com "quem"**: kafka não implementa mecanismos comumente encontrados em ferramentas tradicionais, como o gerenciamento de entrega por consumidor. Esse tipo de controle demanda um processamento adicional e degrada a vazão de dados. Então, temos o controle *offset* por grupos de consumo, que veremos em detalhes adiante.
- **Uma fila**: não, kafka não é uma fila da forma como conhecemos, onde existe exatamente um produtor e exatamente um consumidor realizando operações *push* e *pop*, respectivamente. Ou uma ordenação global dos registros, nem sua eliminação quando todos os consumidos já o receberam.
- **Uma ferramenta "inteligente"**: kafka deixa de lado controles que prejudicam a vazão de dados, deixando-os para o software consumidor e produtor. Só implementando o mínimo, de forma muito inteligente, como veremos em "Funcionamento interno".
- **Lento ou de execução pesada**: Kafka é escrito usando Scala e roda sobre a Java Virtual Machine e não requer uma infraestrutura cara ou proprietária.
- **Guiado por apenas uma empresa**: isso garante que essa ferramenta é guiada pela comunidade de código aberto e não por interesses corporativos.

Ecossistema Apache Kafka

Apache Kafka é um ecossistema de ferramentas rico, criado para atender diversos casos de uso e resolver problemas da computação distribuída de forma clara e objetiva. Nele encontramos a ferramenta certa para cada situação do nosso projeto de transformação digital.

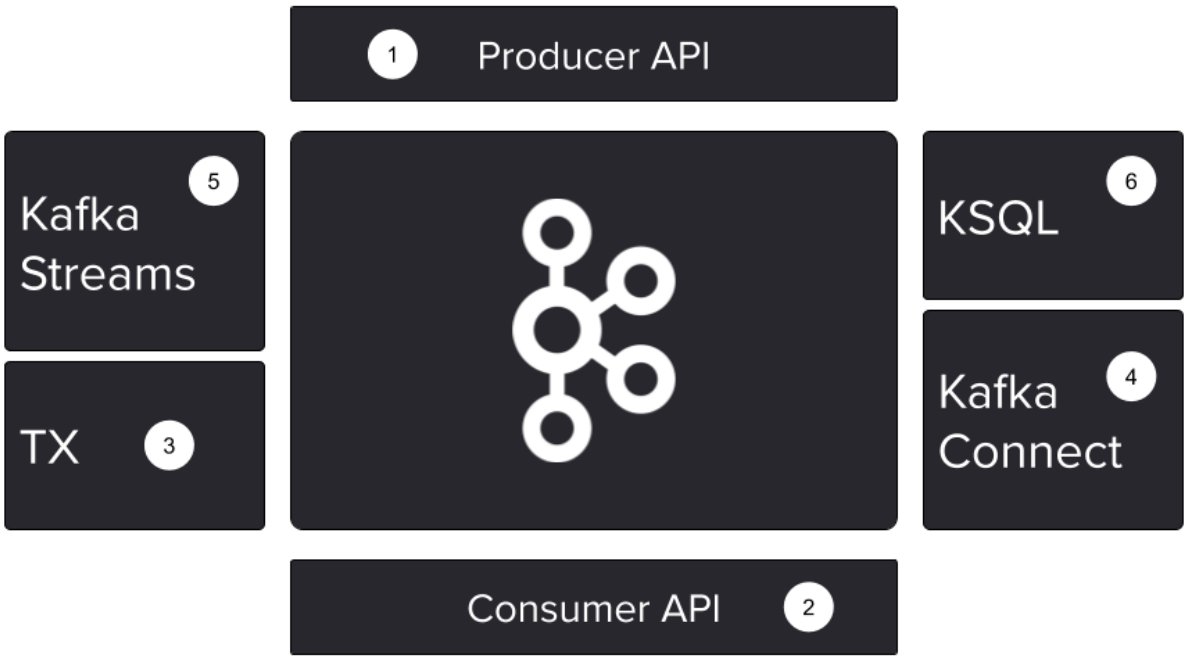

Figura 5: Ecossistema Apache Kafka

1. **Producer API**: API mais básica para produzir eventos no Kafka. Utilizada como base para todas as outras ferramentas.
2. **Consumer API**: API mais básica para consumir eventos do Kafka. Que também é a base para muitas outras implementações.
3. **Transactions**: API para operar comunicação transacional, tanto na produção como no consumo de eventos. Ela também é base para Kafka Streams e Kafka Connect.
4. **Kafka Connect**: Integrar Kafka com origens de dados externas, como bancos de dados, arquivos, APIs, filas tradicionais, etc.
5. **Kafka Streams**: processamento de streams de eventos de forma declarativa no Java.
6. **KSQL**: processar streams de eventos com uma notação SQL like.

Lab 1: configurando Kafka CLI

- Crie o diretório `kafka-m1` no seu computador
  - Se você utiliza o windows, crie o diretório `kafka-m1` na raiz do sistema, ou seja, na unidade `C:\`, ficando assim:

```
C:\kafka-m1
```

- Isso é necessário porque no windows existe uma limitação quanto ao número máximo de caracteres em um caminho de diretórios.
- Realize o download do arquivo `kafka_2.12-2.4.0.zip`, disponível através do link abaixo:

  - http://bit.ly/kafka_2_4_0

- Extraia o conteúdo do arquivo `kafka_2.12-2.4.0.zip` no diretório `kafka-m1`
  - **Linux**: `unzip kafka_2.12-2.4.0.zip`
  - **Windows**: *utilize o aplicativo 7-zip ou similar*
- Configure a variável de ambiente `KAFKA`.
  - **Linux**: `export KAFKA=/path/to/kafka-m1/kafka_2.12-2.4.0`
- Configure a variável de ambiente `PATH`, assim você poderá executar os comandos Kafka a partir de qualquer local do seu computador.
  - **Linux**: `export PATH=$PATH:$KAFKA/bin`

Windows

Guarde o caminho onde você extraiu o conteúdo do arquivo `kafka_2.12-2.4.0.zip`:

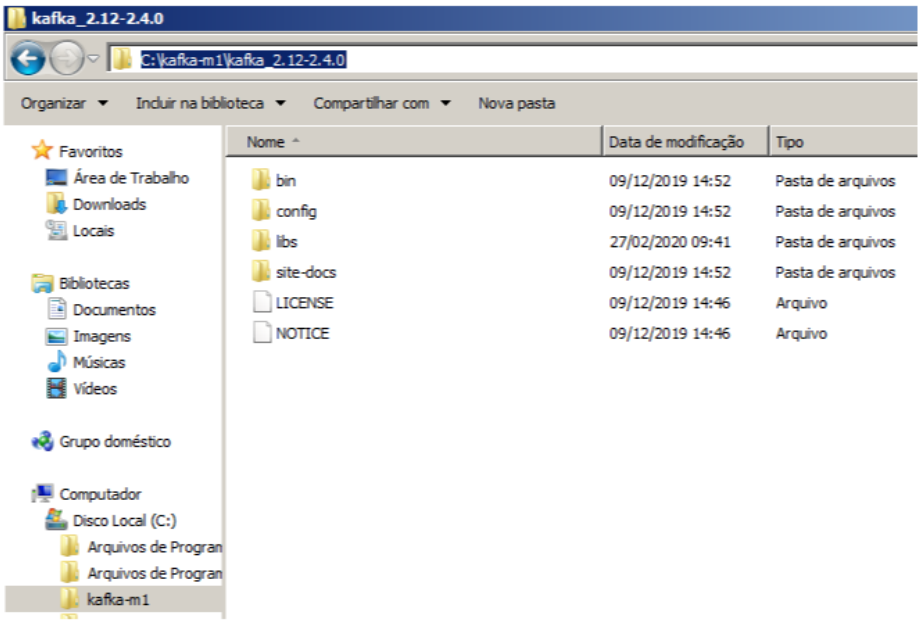

Figura 6: Exemplo de caminho para instalação kafka

Procure no menu iniciar por **Editar as variáveis**:

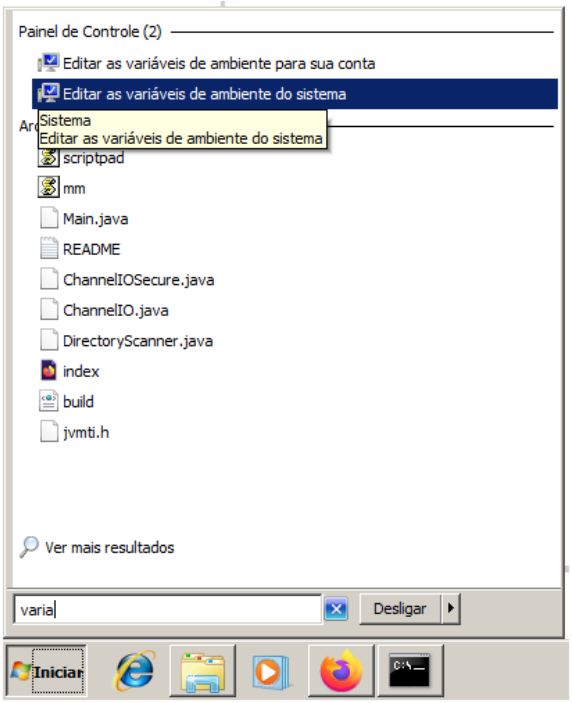

Figura 7: Menu iniciar no windows

Clique no botão **Variáveis de Ambiente...**:

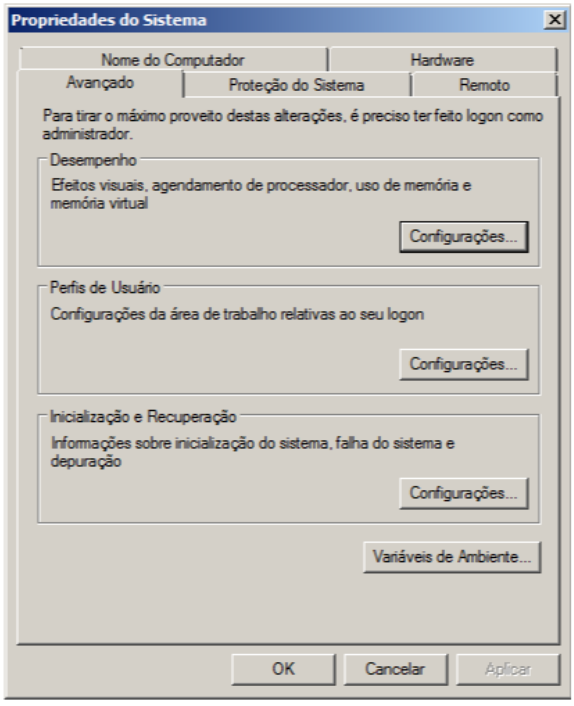

Figura 8: Propriedades de sistema no windows

Clique no botão **Novo...**:

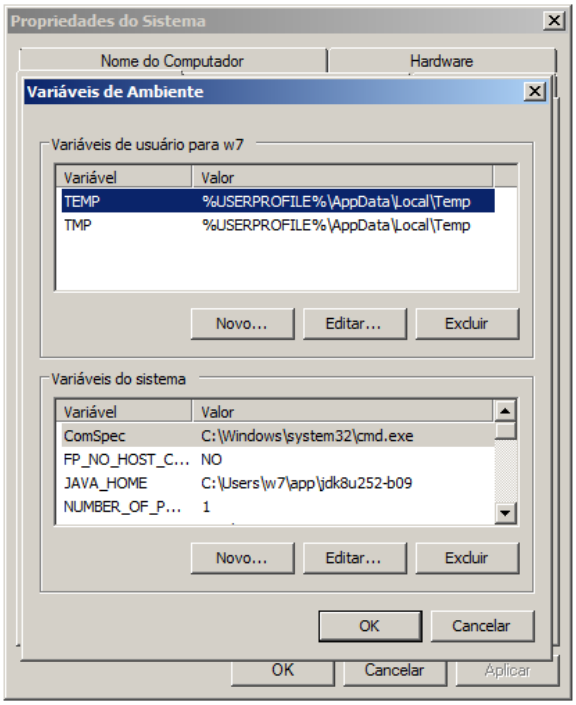

Figura 9: Variáveis de ambiente

Crie uma variável chamada `KAFKA` com o valor `C:\kafka-m1\kafka_2.12-2.4.0`

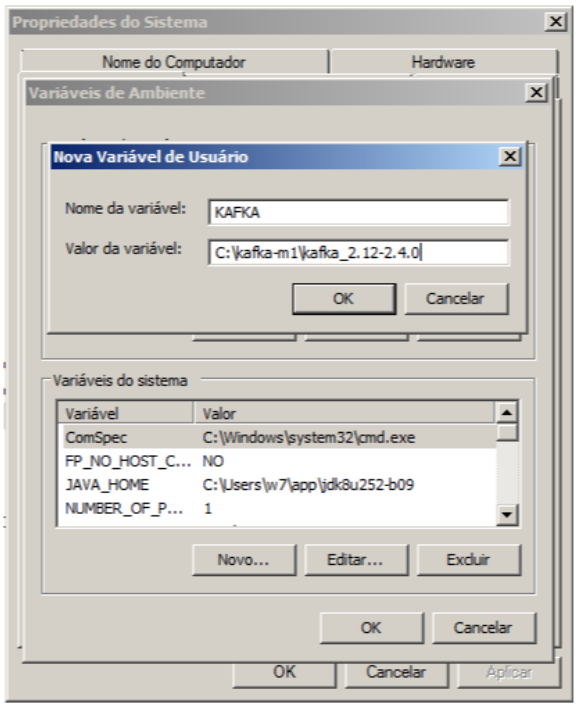

Figura 10: Nova variável KAFKA

Crie ou atualize uma variável chamada `PATH`, com o valor `%PATH%;%KAFKA%\bin\windows`

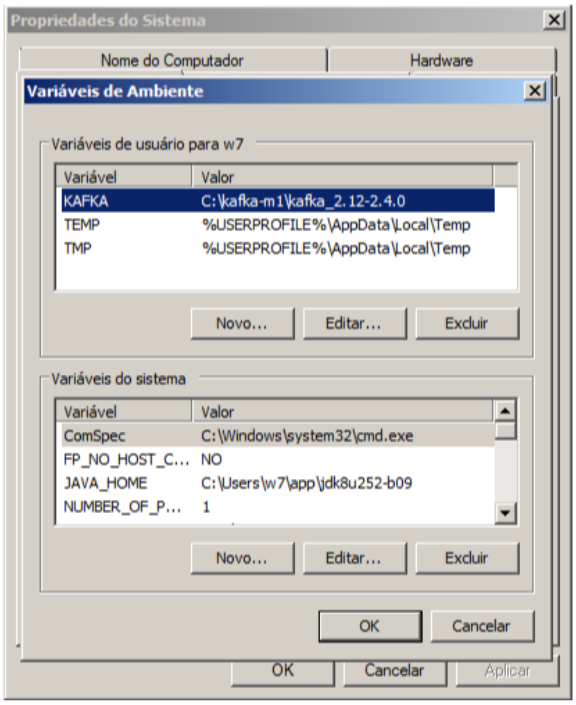

Figura 11: Nova variável PATH

Depois de criá-las, confirme tudo clicando em **OK**:

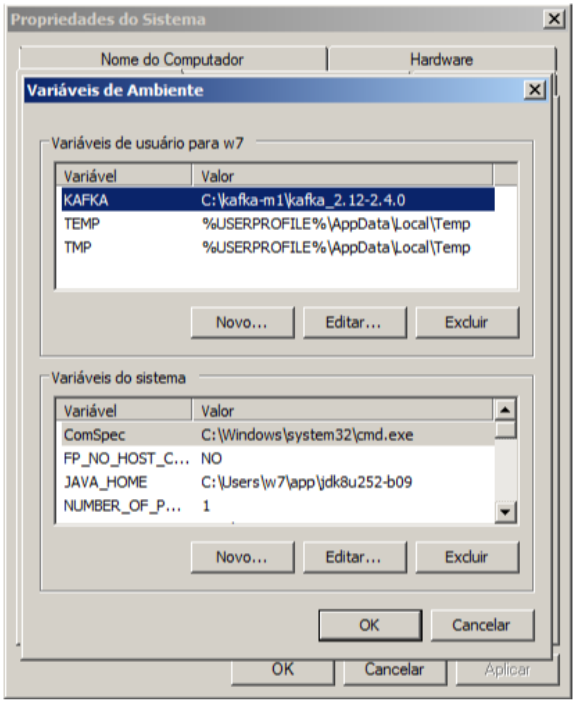

Figura 12: Variável KAFKA criada

Cluster Kafka para Testes

Temos um cluster kafka compartilhado e ele estará funcionando durante todo o curso. Você poderá utilizá-lo em todos os laboratórios que serão executados e no Apêndice I existem detalhes sobre como executar o mesmo cluster na sua máquina.

Estes são os endereços para utilizar na configuração `bootstrap.servers`:

```
b0.kafka.ml:9092,b1.kafka.ml:9092
```

> É muito importante ter um cluster kafka em seu PC para seus estudos ou testes posteriores.

Infelizmente, se você utiliza Windows, não será uma tarefa simples executar um cluster kafka e utilizar todas as suas funcionalidades. A não ser que você tenha Docker instalado.

Meu primeiro tópico

Agora que temos um cluster funcional, vamos criar nosso primeiro tópico e descrever seus detalhes para entender como é sua operação.

- Abra um novo terminal **Linux**:
  - Digite o comando:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 7 \
  --topic WB-meu.topico
```

- Abra o prompt de comando **Windows**:
  - Digite o comando:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 7 ^
--topic WB-meu.topico
```

`kafka-topics.sh` ou `kafka-topics.bat`, são comandos para realizar operações relacionadas aos tópicos, como criar, listar, apagar ou detalhar suas configurações. Os argumentos utilizados foram:

- `--bootstrap-server` Ponto de contato com o cluster Kafka, que deverá ser `b0.kafka.ml:9092,b1.kafka.ml:9`
  ou aquele servidor de contingência que disponibilizamos.
- `--replication-factor` Número de réplicas deste novo tópico.
- `--partitions` Número de partições dentro deste novo tópico.

> Mais detalhes na seção Anatomia dos tópicos.

Depois de criar nosso tópico, vamos detalhar seus atributos para entender como ele está dentro do cluster e como foram distribuídos o líder-réplicas, ou *leader-followers* em inglês.

Linux

```
kafka-topics.sh --describe \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-meu.topico
```

Windows

```
kafka-topics.bat --describe ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-meu.topico
```

Exemplo de saída:

```
Topic: meu.topico    PartitionCount: 7    ReplicationFactor: 3
Configs: segment.bytes=1073741824
    Topic: meu.topico    Partition: 0    Leader: 1    Replicas: 1,2,3    Isr: 1,2,3
    Topic: meu.topico    Partition: 1    Leader: 2    Replicas: 2,3,1    Isr: 2,3,1
    Topic: meu.topico    Partition: 2    Leader: 3    Replicas: 3,1,2    Isr: 3,1,2
    Topic: meu.topico    Partition: 3    Leader: 1    Replicas: 1,3,2    Isr: 1,3,2
    Topic: meu.topico    Partition: 4    Leader: 2    Replicas: 2,1,3    Isr: 2,1,3
    Topic: meu.topico    Partition: 5    Leader: 3    Replicas: 3,2,1    Isr: 3,2,1
    Topic: meu.topico    Partition: 6    Leader: 1    Replicas: 1,2,3    Isr: 1,2,3
```

- Cada partição é listada, de `0` a `6`.
- `Topic`: nome do tópico.
- `PartitionCount`: número total de partições.
- `ReplicationFactor`: fator de replicação das partições deste tópico. Esse valor define o número de réplicas que o kafka tentará manter em sincronia.
- `Configs`: configurações informadas no argumento `--config` ao criar o tópico. Todas as outras são valores-padrão definidos na configuração do servidor e que foram herdadas por nosso tópico.

Funcionamento interno

Vamos a uma breve introdução sobre como kafka opera internamente os tópicos, em especial a replicação das partições no cluster.

- **Leader**: o broker onde reside a partição líder da replicação, ela é a responsável por determinar quais outras réplicas estão sincronizadas. E é somente nela que os dados serão produzidos e consumidos. Uma característica importante, é que jamais `Leader` e `Replica` estarão no mesmo broker por questões de alta-disponibilidade.

- **Replicas**: são os backups das partições, suas cópias, também referenciadas como *Followers*. Elas, naturalmente residem em outros brokers do cluster e entram em ação no momento em que o `Leader` falha, sendo uma delas eleita como a nova líder.
- **Isr** - *in-sync replicas*: conjunto de réplicas que estão sincronizadas com a `Leader`. Um exemplo de seu uso, é quando trabalhamos com transações: elas somente são efetivadas (commit) quando todas as `Isr` retornarem ACK para a replicação dos registros.

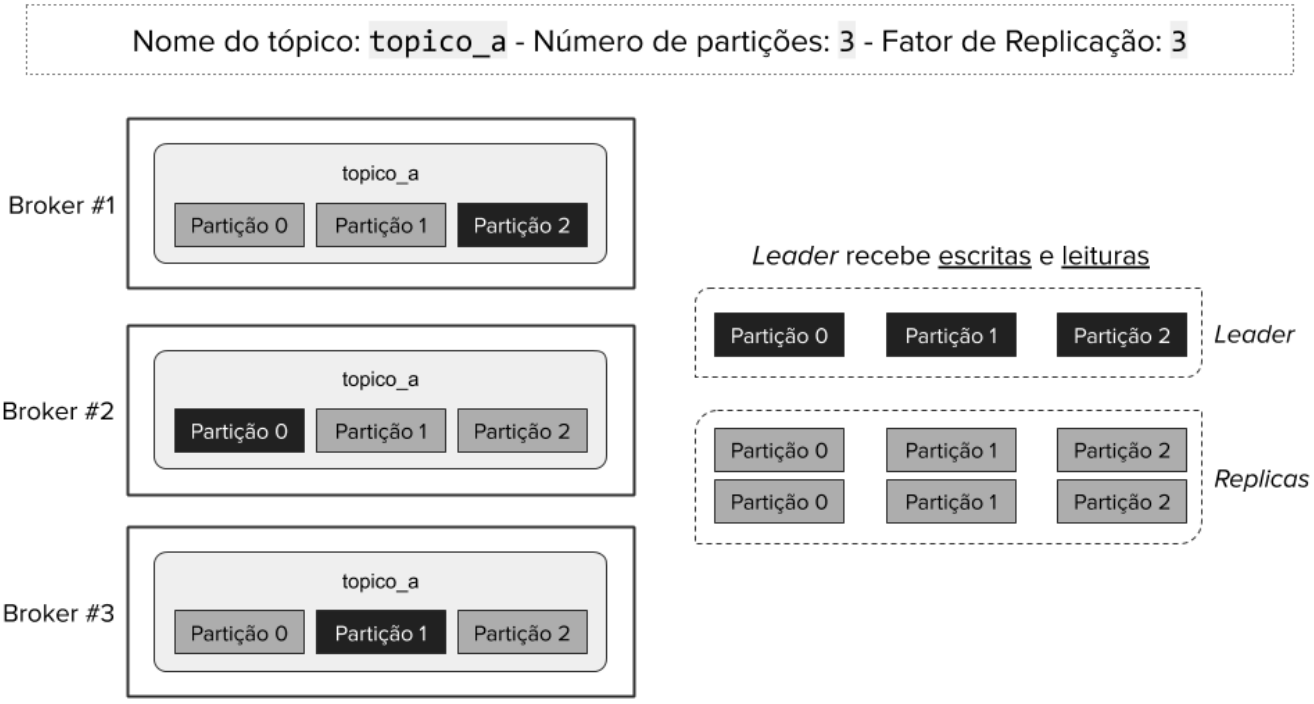

Figura 13: Líder-réplicas das partições

Anatomia dos Tópicos

Tópicos são divisões lógicas para armazenamento de dados, pois são em suas partições, as divisões físicas, onde realmente os dados seguem persistentes e duráveis. Quanto ao número máximo de tópicos, virtualmente não existe uma limitação para criá-los no cluster kafka.

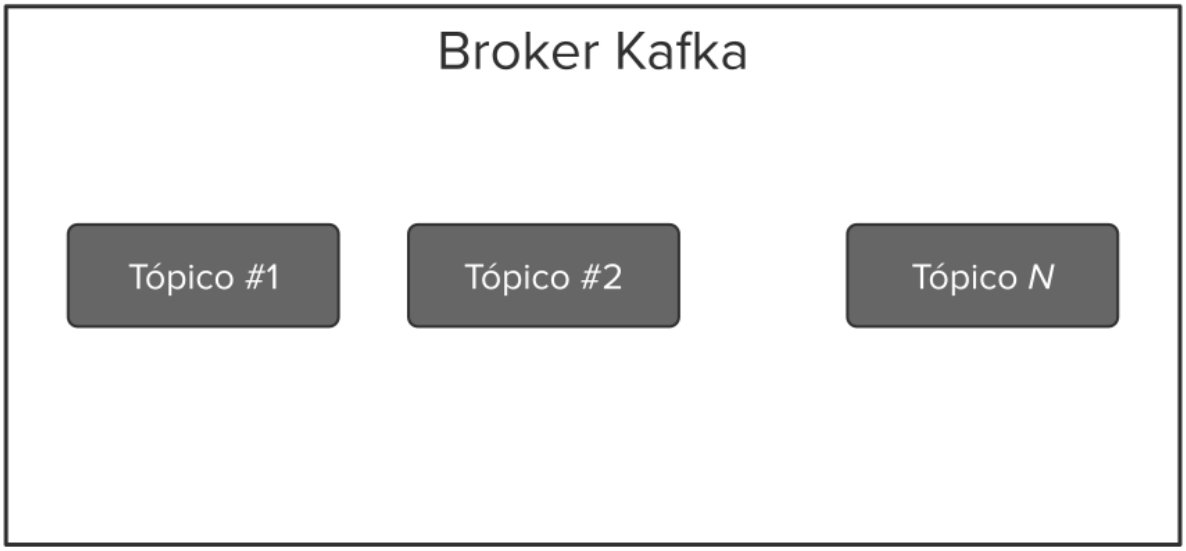

Figura 14: Em cada broker temos vários tópicos

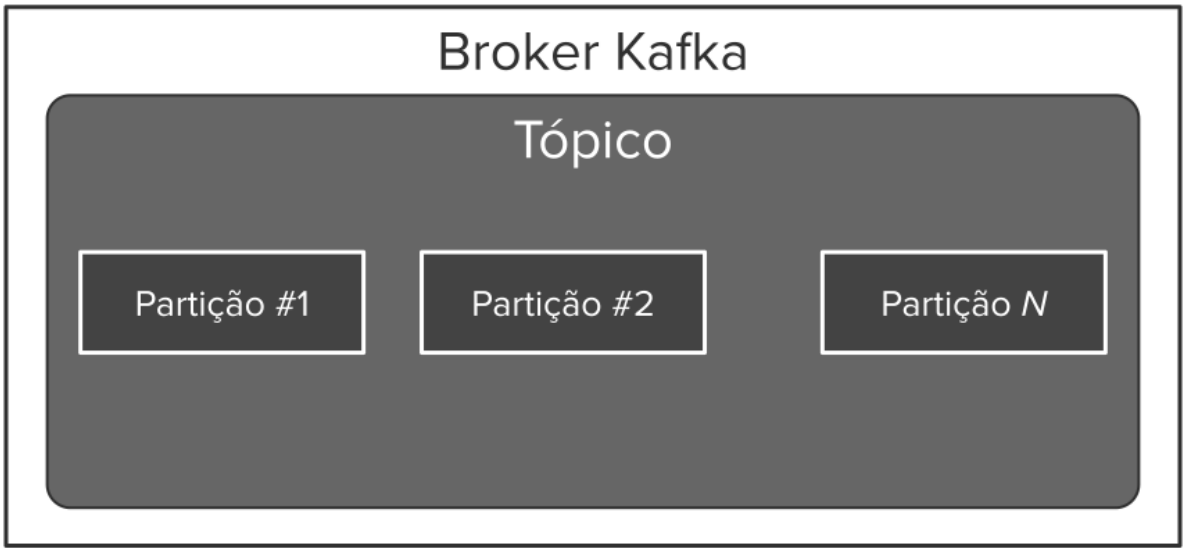

Figura 15: Em cada tópico existem várias partições

As partições persistem fisicamente os dados dentro de um tópico e proporcionam paralelismo em seu consumo.

- tecnicamente, as **partições** são comumente referenciadas como **unidades de paralelismo**

para o consumo.

O número de partições por tópico é virtualmente ilimitado. E um número elevado de partições aumenta o paralelismo, mas pode causar lentidão na replicação e no re-balanceamento de consumidores ou no *failover*. Mas é comum observar instalações com duas ou cinco mil partições em produção, rodando perfeitamente.

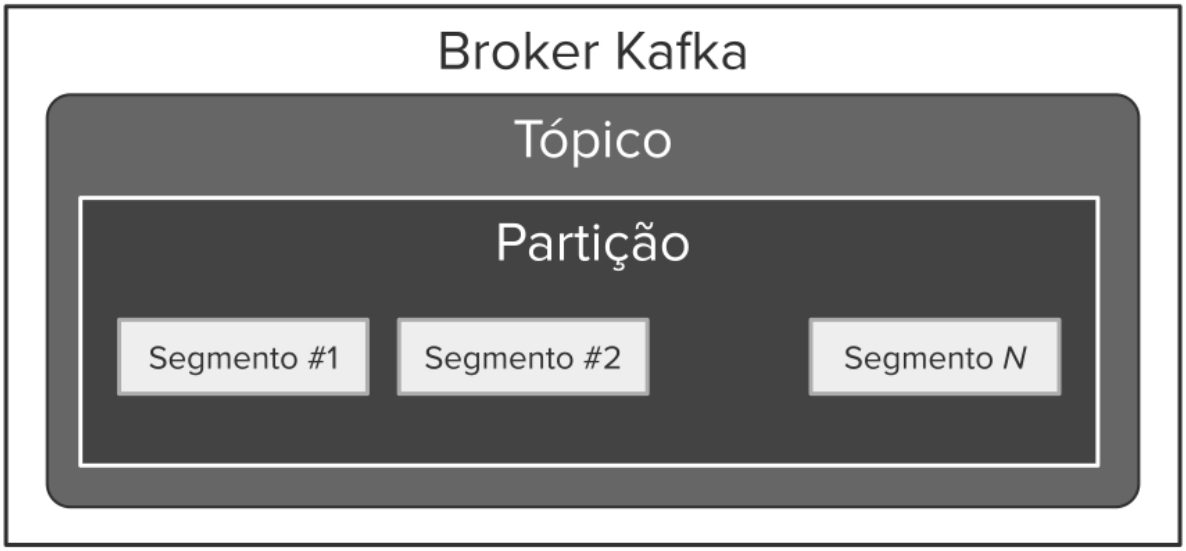

Figura 16: Menor unidade de armazenamento, os segmentos

Então chegamos a menor unidade de armazenamento de dados no Kafka:

- os **segmentos**

Segmentos são os arquivos físicos com um tamanho limitado e que armazenam os registros das partições. Com esse desenho, gravar e ler dados se torna eficiente.

Kafka não permite busca por chave de registro, pois ele foi desenhado para alta vazão de dados, então o acesso acontece através de *offset* e da leitura em série. Os *offsets* são como um tipo de ponteiro que são mapeados para as posições físicas nos segmentos onde residem os dados.

> Também é possivel encontrar o **offset** com base em um carimbo data-hora, o timestamp.

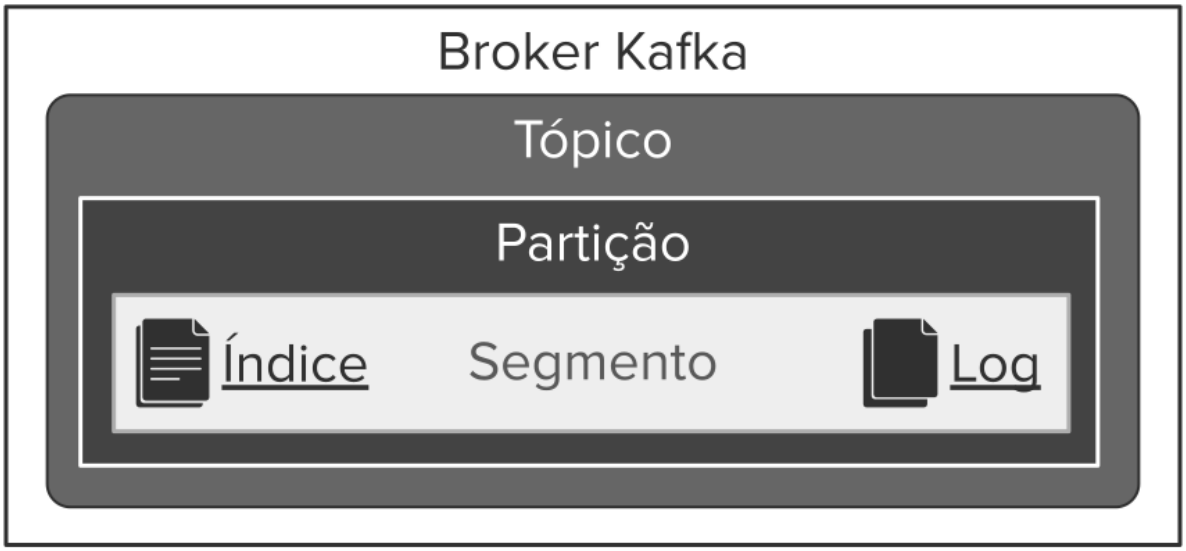

Figura 17: Segmento é composto pelos arquivos de índice e de log

Cada **segmento** é composto por tres arquivos principais:

- **índice para offset**: mapeamento entre *offset* e a posição física no arquivo de log.
- **índice para timestamp**: mapeamento entre carimbo data-hora e *offset*.
- **log**: arquivo físico onde estão os dados ou lotes de registros.

Kafka ao criar arquivos de segmento, os nomeia com o *offset* inicial neles armazenados. Vamos ilustrar isso com um exemplo:

- Imagine os *offsets* de `0` a `10`. Eles estarão presentes nos arquivos:
  - `00000000000000000000.index`
  - `00000000000000000000.log`
- Já os *offsets* de `11` em diante, estarão nos arquivos:
  - `00000000000000000011.index`
  - `00000000000000000011.log`

Assim ao ler dados do armazenamento para enviar aos consumidores, existirá o acesso direto ao arquivo de índice para obter a posição física no log onde o *offset* se inicia. Então ele acessa o arquivo de log e faz a leitura em série.

Exemplo:

- Imagine que o consumer solicitou dados ao kafka, e digamos que o último *offset* confirmado no seu grupo de consumo foi o de número `13`.
- O broker então, vai até o seguinte arquivo de índice:
  - `00000000000000000011.index`

- Então, o broker lê no arquivo de índice qual a posição física onde reside o *offset* 13
- Com a posição física, o broker inicia a leitura em série do arquivo de log:
  - `00000000000000000011.log`

A criação de segmentos é ditada por duas configurações feitas nos tópicos:

- `segment.bytes`: tamanho máximo em bytes para cada segmento.
- `segment.ms`: tempo para que o kafka crie um novo segmento mesmo que o atual ainda não tenha atingido seu tamanho máximo. Essa configuração existe para que aconteça a compactação ou deleção, pois enquanto o segmento estiver aberto não é possível processá-lo.

Dentre os segmentos presentes em uma partição, sempre haverá apenas um ativo. E este segmento que recebe todos os registros que estão chegando dos produtores. E um detalhe muito importante é que segmentos ativos não são elegíveis para processo de limpeza e compactação.

Compactação

A compactação de log, *log compaction* em inglês, é um procedimento operacional no Kafka que tem o objetivo de reduzir o tamanho dos dados persistidos. Ele é ativado quando se utiliza o valor `compact` na configuração `cleanup.policy`.

Primeiro vejamos a produção de registros dentro do kafka e sua estruturação.

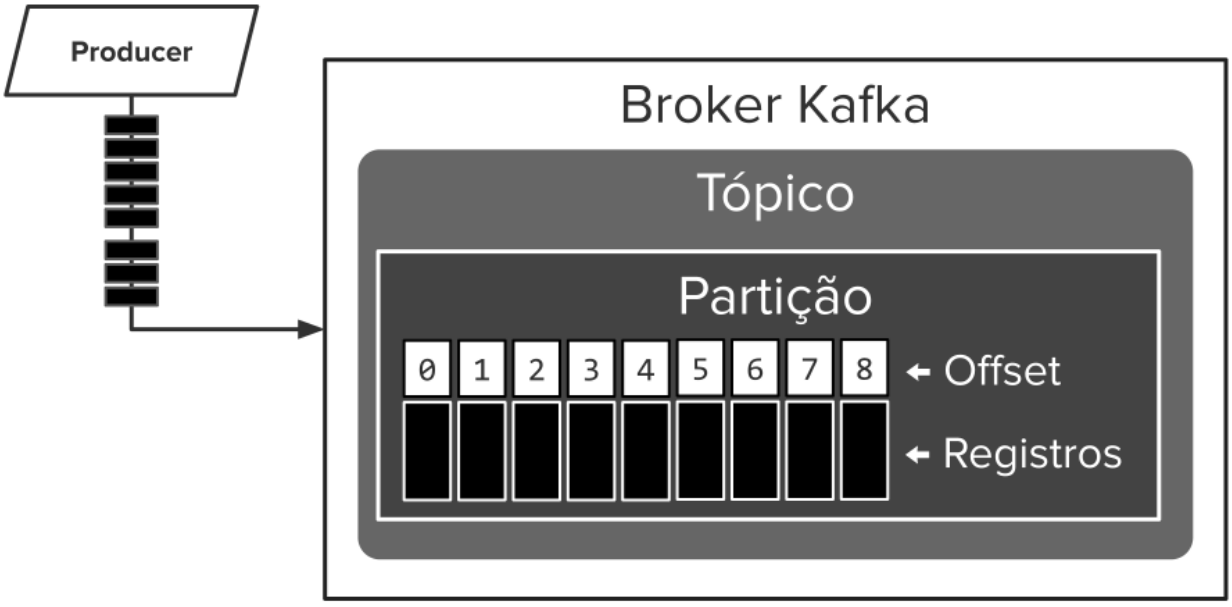

Figura 18: Produção de registros recebendo offsets nas partições

Registros são produzidos, seguindo para as partições do tópico e recebendo um *offset* toda vez que são escritos no final do segmento ativo. Pensemos que os *Offsets* são uma espécie de índice dos registros.

> Veremos producers em detalhes na seção Produtores.

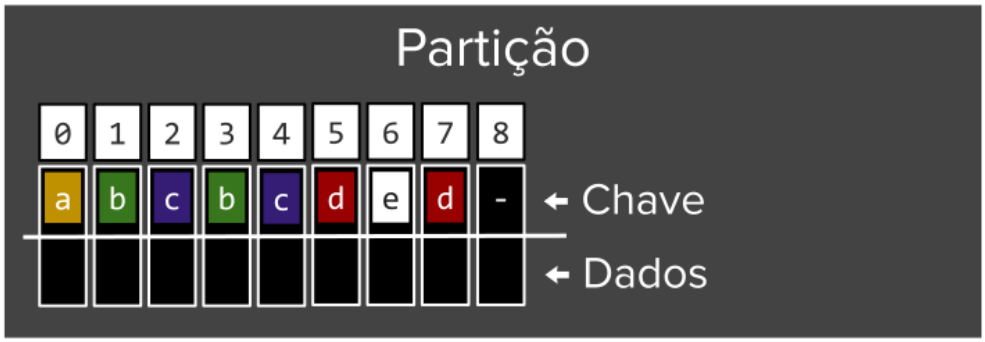

Figura 19: Registros possuem chaves, como: a, b, c, d, e

Cada registro pode ser produzido com uma chave, key em inglês, que é utilizada para determinar em qual partição ele será persistido. E kafka garante, na configuração padrão, que registros com a mesma chave sigam sempre para a mesma partição.

Também é possível notar que alguns registros não possuem chave, como o registro no *offset* número 8. Nesses casos haverá uma distribuição entre as partições disponíveis, um tipo de distribuição de carga, *load balancing* em inglês.

> Veremos registros em detalhes na seção Anatomia dos registros.

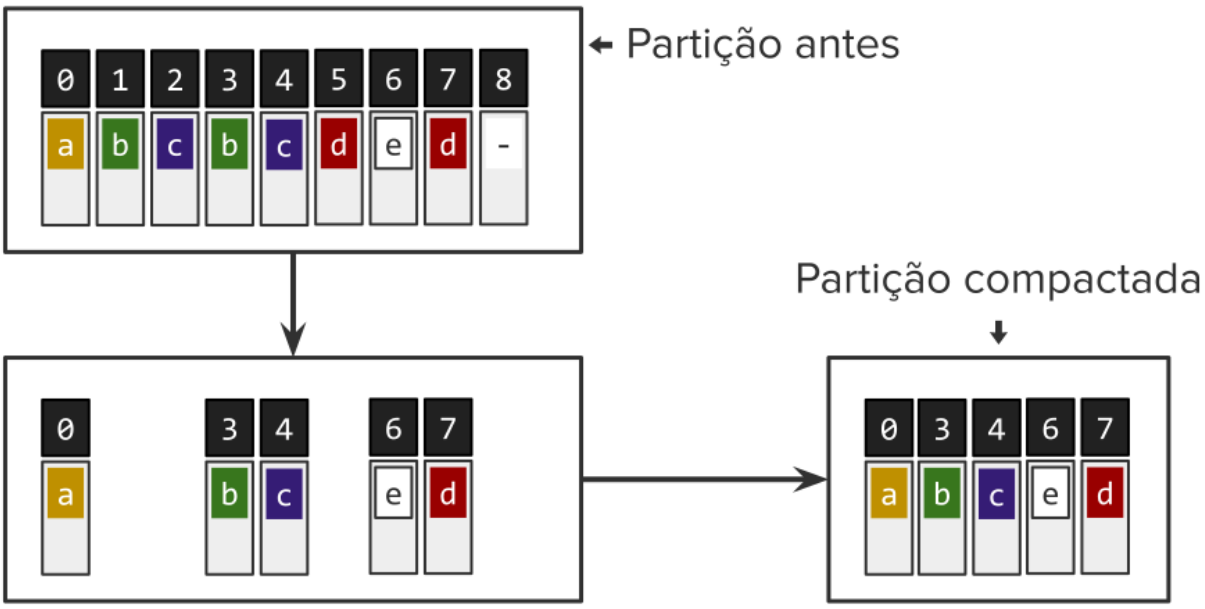

Figura 20: Partição após compactação

A compactação de log é executada de forma automatizada pelo Kafka, eliminando registros antigos

que possuam outro mais novo e com chave idêntica. Já os registros que não possuem chave, como aquele no *offset* número 8, são completamente eliminados do log.

Durante a compactação, os registros são marcados para eliminação, *delete markers* em inglês. Já os *offsets*, são mantidos, como observamos na figura anterior:

- os registros nos *offsets* `1` e `2` foram eliminados, mas nenhuma reordenação ou novos *offsets* foram atribuídos.

Manter os *offsets* é importante porque caso contrário, todos os segmentos deverão ser completamente re-gerados degradando o desempenho geral dos brokers e sua taxa de transferência, *throughput* em inglês.

Principais configurações

Todas as configurações de um tópico possuem valores-padrão definidas no broker, através do arquivo `config/server.properties`. E quando forem criados herdarão essas configurações, desde que não sejam sobrescritas através do argumento `--config` do comando `kafka-topics.sh` (ou `kafka-topics.bat` para windows). Ou que por ventura forem criados automaticamente.

Existem cerca de 26 configurações para um tópico, vejamos então àquelas cruciais para nosso dia-a-dia no desenvolvimento de projetos.

> Confira a documentação oficial para conhecer todas as outras configurações.

cleanup.policy

Define como será o processo de limpeza dos tópicos.

Para manter os dados sanitizados nos tópicos, o Kafka implementa sua limpeza, seja pela eliminação de registro ou pela remoção das repetições como a compactação.

- valor padrão: `delete`
- valores possíveis
  - `delete`: após o tempo ou tamanho máximos serem atingidos, os segmentos são limpos e eliminados fisicamente do armazenamento.
  - `compact`: serão mantidos somente os dados mais recentes sob uma determinada chave, já os antigos, são apagados através da compactação de log. Com essa configuração, qualquer registro que não contenha chave, será rejeitado para persistência no tópico, respondendo erros para o *producer* e não persistindo-o (KIP-135).
  - `delete,compact`: além da compactação de log, também será possível reduzir o tamanho do espaço ocupado através da eliminação dos segmentos antigos.
- configurações relacionadas
  - `min.compaction.lag.ms` quando valor é `compact`

  - `max.compaction.lag.ms` quando valor é `compact`
  - `retention.bytes` quando o valor é `delete`
  - `retention.ms` quando o valor é `delete`
  - `min.cleanable.dirty.ratio` quando o valor é `compact`
  - `segment.bytes`
  - `segment.ms`

- configuração no broker p/ valor padrão: `log.cleanup.policy`

A limpeza que apaga os dados, delete, é realizada sobre os segmentos, ou seja, não é uma política aplicada individualmente aos registros. Kafka só processa segmentos inativos quando o carimbo data-hora mais antigo neles existente, expirar.

Podemos habilitar ambas as políticas e configurá-las, deste modo:

```
cleanup.policy=compact,delete
```

compression.type

Designar um tipo de compressão para registros que são persistidos no tópico.

- valor padrão: `producer`
- valores possíveis:
  - `uncompressed`: nenhum tipo de compressão.
  - `zstd`: algoritmo de compressão criado pelo Facebook.
  - `lz4`: valor recomendado, quando você utilizar compressão.
  - `snappy`: algoritmo de compressão criado pelo Google.
  - `gzip`: compressão zip tradicional.
  - `producer`: compressão criada pelo producer é mantida, persistida no tópico e enviada aos consumidores.
- configuração no broker p/ valor padrão: `compression.type`

delete.retention.ms

Tempo total em que serão mantidos os marcadores de eliminação, *delete markers* em inglês ou também *tombstones*. Essa configuração delimita o tempo total em que os consumidores devem completar a leitura dos dados do tópico antes que os dados sejam eliminados fisicamente.

> A remoção do tombstone e leitura dos segmentos acontecem simultaneamente.

Essa configuração garante que registros marcados para eliminação serão mantidos o tempo suficiente para que os consumidores tenham tempo para lê-los antes da eliminação definitiva, ou seja, quando elegível para eliminação, o registro ainda permanecerá 24h no log do tópico, caso a configuração padrão seja mantida.

Apesar de remeter ao *delete*, essa configuração só é aplicada quando o `cleanup.policy` for igual a `compact`.

- valor padrão: `86400000` (24h)
  - configure `0` para desativá-la
- configurações relacionadas
  - `cleanup.policy` com valor igual a `compact`
- configuração no broker p/ valor padrão: `log.cleaner.delete.retention.ms`

max.compaction.lag.ms

Tempo máximo em que os dados são inelegíveis para compactação, ou seja, esse tempo começa a contar à partir do momento em que eles são persistidos no tópico e este é o máximo de tempo em que eles permaneceram no estado em que foram persistidos.

- valor padrão: capacidade máxima do tipo *long* que é 263-1
- configurações relacionadas
  - `cleanup.policy` com valor igual a `compact`
- configuração no broker p/ valor padrão: `log.cleaner.max.compaction.lag.ms`

min.compaction.lag.ms

Tempo mínimo em que os dados permanecem sem passar pela compactação, ou seja, esse tempo começa a contar à partir do momento em que eles são persistidos no tópico.

A relação entre esta configuração e a `max.compaction.lag.ms`, é que na max definimos o tempo máximo em que um dado permanecerá no tópico sem passar pela compactação, ou seja, quando aquele tempo for atingido é garantido que os dados foram compactados.

Caso o segmento possua um registro com carimbo data-hora, *timestamp* em inglês, mais novo que este valor, ele, o segmento, não passará pelo processo de compactação. Mas isso só é válido se configurarmos um valor superior à zero.

- valor padrão: `0`
- configurações relacionadas
  - `cleanup.policy` com valor igual a `compact`
- configuração no broker p/ valor padrão: `log.cleaner.min.compaction.lag.ms`

max.message.bytes

Tamanho máximo permitido para uma mensagem.

Caso o lote de registros do producer seja maior que o valor desta configuração, sua requisição será rejeitada. O erro recebido será algo similar a este:

```
The request included a message larger than the max message size the server will accept.
```

Ou dependendo da linguagem ou biblioteca de conexão com o Kafka, a seguinte mensagem:

```
Message size too large
```

Todas elas levarão a entender que a mensagem é maior que o valor definido nesta configuração.

> Sempre devemos entender que mensagem trata-se na verdade de um lote, batch em inglês, de registros.

- valor padrão: `1000012` (~976 Kb)
- configurações relacionadas
  - `batch.size`, uma configuração no producer
- configuração no broker p/ valor padrão: `message.max.bytes`

message.timestamp.type

Qual o tipo de estampa para data-hora será utilizada para identificar quando a mensagem foi criada. Diversas operações do kafka sobre os dados de um tópico tem como base o carimbo data-hora, para definir quanto tempo de vida ele tem, por exemplo.

- valor padrão: `CreateTime`
- valores possíveis:
  - `CreateTime`: carimbo data-hora enviados pela aplicação que produziu a mensagem.
  - `LogAppendTime`: carimbo data-hora do broker kafka que sobrescreve àquele proveniente da aplicação produtora.
- configurações relacionadas:
  - `message.timestamp.difference.max.ms` quando se utiliza `CreateTime`
- configuração no broker p/ valor padrão: `log.message.timestamp.type`

retention.bytes

Tamanho máximo que a partição poderá atingir, com todos os seus segmentos, até que se torne elegível para o processo de eliminação dos registros antigos.

- valor padrão: `-1` (desativado)
  - configure `-1` para desativá-la
- configurações relacionadas:
  - `cleanup.policy` com valor igual a `delete`
- configuração no broker p/ valor padrão: `log.retention.bytes`

Se esta configuração estiver ativa junto da `retention.ms`, a limpeza será desencadeada por aquela que atingir o limite primeiro.

retention.ms

Tempo máximo em que os **segmentos** serão mantidos, iniciando a contagem a partir de sua produção com base no carimbo data-hora.

- valor padrão: `604800000` (7 dias)
  - configure `-1` para desativá-la (tempo máximo indefinido)
- configurações relacionadas:
  - `cleanup.policy` com valor igual a `delete`
- configuração no broker p/ valor padrão: `log.retention.ms`

Se esta configuração estiver ativa junto da `retention.bytes`, a limpeza será desencadeada por aquela que atingir primeiro o limite.

O processamento da retenção baseada em tempo é feita somente quando o carimbo data-hora do registro mais recente expirar, ou seja, todos os outros, mais antigos, ainda estarão disponíveis para consumo até que o segmento seja limpo. Por este motivo existe o tamanho máximo dos segmentos, controlado pela configuração `segment.bytes`.

segment.bytes

Define o tamanho máximo, em bytes, para os arquivos de segmento. Toda vez que o arquivo atual atingir este tamanho, um novo segmento é criado e torna-se o segmento ativo. Mas isso não acontece através de um processo automatizado que varre os arquivos de segmento buscando elegíveis. Essa operação é desencadeada quando novos registros são produzidos, então o broker identifica que esse tamanho máximo foi atingido e cria um novo segmento.

- valor padrão: `1073741824` (~1GB)
  - configuração no broker p/ valor padrão: `log.segment.ms`

A criação de um novo segmento está condicionada a esta configuração e a `segment.ms`, sendo desencadeada por àquela que atingir o primeiro o limite.

segment.ms

Tempo máximo que o kafka aguardará até forçar a criação de um novo segmento. Mas isso não acontece através de um processo automatizado que varre os segmentos em busca daqueles elegíveis, essa operação é desencadeada quando novos registros são produzidos e o broker identifica que esse tempo foi atingido e cria um novo segmento. .

- valor padrão: `604800000` (7 dias)
  - valor mínimo: `1`
- configuração no broker p/ valor padrão: `log.roll.ms`

A criação de um novo segmento está condicionada a esta configuração e a `segment.bytes`, sendo desencadeada por àquela que atingir primeiro o limite.

> Confira a documentação oficial para conhecer todas as outras configurações.

min.insync.replicas

Número mínimo de réplicas que deverão responder sobre a sincronização. Ela é uma configuração importante quando o Producer estiver configurado com `acks=all`, determinando quantas réplicas, além do líder, responderam sobre o conhecimento de entrega do lote de registros produzido.

- valor padrão: `1`
- configurações relacionadas
  - `acks` no Producer

Por exemplo, com esta configuração igual a `2` e `acks=all`, o producer irá receber ao todo três confirmações: uma do líder e outras duas uma de cada réplica. Porém, se não houverem réplicas suficientes para manter a sincronização será lançada a exceção NotEnoughReplicas para o cliente que produziu os registros.

min.cleanable.dirty.ratio

Determinar com qual frequência os segmentos das partições do tópico serão processados pela compactação de log. Quanto maior for este valor, menor será a frequência da compactação.

- valor padrão: `0.5`
- configurações relacionadas
  - `max.compaction.lag.ms`
  - `min.compaction.lag.ms`

O processo interno do Kafka para limpeza de tópicos calcula qual é a proporção de duplicatas dentro dos segmentos inativos, que por padrão devem ser de `50% (0.5)` ou mais. E se por exemplo definirmos `10% (0.1)`, isto significa que a se houverem apenas `10%` de duplicatas, o segmento é elegível para compactação.

Configurações para operação

Existem três configurações especiais nos tópicos que serão utilizadas no instante de sua criação e determinam como será sua operação dentro do cluster por exemplo.

- `partitions`: número de partições disponíveis no tópico
  - Este valor pode ser modificado após criá-lo, mas nenhuma redistribuição de registros é realizada. Portanto isso deve ser realizado com cautela somente em casos especiais.
  - Pense neste valor à frente, pois é ele quem ditará o paralelismo máximo no consumo.
  - Utilize este artigo como base para determinar o valor dessa característica
- `replication-factor`: Número de réplicas, ou cópias, deste novo tópico dentro do cluster Kafka. Que sempre deverá ser inferior ou igual ao número total de brokers.
  - Este valor define o número de cópias que o kafka tentará manter em sincronia.
- `name`: nome do tópico
  - Não são permitidos nomes idênticos
  - Este é o identificador do tópico dentro do cluster kafka

Lab 2: Criando tópicos

Agora vamos exercitar a criação de tópicos com configurações variadas e entender seu comportamento, com os resultados dos comandos.

> Todas as configurações que são para producers ou consumers, serão vistas em detalhes nas seções Produtores e Consumidores.

Experimento a

Vamos criar o tópico `WB-lote300kb`, que restringe o tamanho máximo do lote de registros através da configuração `max.message.bytes`.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 7 \
  --topic WB-lote300kb \
  --config max.message.bytes=307200
```


Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 7 ^
--topic WB-lote300kb ^
--config max.message.bytes=307200
```

- `307200` é o equivalente a `300KB`

Produza lotes de registros **obedecendo** à limitação.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-lote300kb \
  --num-records 5 \
  --record-size 51200 \
  --throughput -1 \
  --producer-props \
    acks=1 \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
    batch.size=307200
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-lote300kb ^
--num-records 5 ^
--record-size 51200 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
batch.size=307200
```

- `51200` é o equivalente a `50KB`
- `307200` é o equivalente a `50KB * 6`

> As configurações no argumento `--producer-props` serão detalhadas na seção Produtores.

A saída dos comandos acima serão algo similar a este:

```
5 records sent, 12.376238 records/sec (0.60 MB/sec), 147.20 ms avg latency, 392.00 ms max
    latency, 86 ms 50th, 392 ms 95th, 392 ms 99th, 392 ms 99.9th.
```

Produza lotes de registros **maiores** que o limite. Ao tentar enviar lotes maiores que o valor definido pela configuração `max.message.bytes`, serão retornados erros e nenhum registro será persistido no tópico.

Erros como este. Mas não se engane, estamos falando do lote, não de um único registro.


```
MESSAGE_TOO_LARGE
```

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-lote300kb \
  --num-records 7 \
  --record-size 51200 \
  --throughput -1 \
  --producer-props \
    acks=1 \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
    batch.size=409600
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-lote300kb ^
--num-records 7 ^
--record-size 51200 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
batch.size=409600
```

- `51200` é o equivalente a `50KB`
- `409600` é o equivalente a `50KB * 8`

Experimento b

Agora crie o tópico `WB-topico2min` com retenção de no mínimo dois minutos.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-topico2min \
  --config cleanup.policy=delete \
  --config retention.ms=120000 \
  --config segment.ms=3000
```

Windows:


```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-topico2min ^
--config cleanup.policy=delete ^
--config retention.ms=120000 ^
--config segment.ms=3000
```

- `120000` é o equivalente a 2 minutos
- `segment.ms` também foi configurado com `3000` (3s), portanto a cada três segundos um novo segmento será forçado a ser criado. Somente assim conseguiremos visualizar tudo funcionando.

Vamos produzir registros no tópico `WB-topico2min`, para então observar seu comportamento.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-topico2min
> [Digite qualquer texto e tecle ENTER]
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-topico2min
> [Digite qualquer texto e tecle ENTER]
```

Para facilitar a visualização do tempo decorrido inclua no texto do registros a hora atual. Digamos que agora seja `9:33`, então podemos fazer o seguinte:

```
> Meu Registro 9:33 - 1
> Meu Registro 9:33 - 2
> Meu Registro 9:33 - 3
```

Liste os registros para certificar que eles estão lá.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-topico2min \
  --property print.key=true \
  --property print.timestamp=true \
  --from-beginning
```

Windows:


```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-topico2min ^
--property print.key=true ^
--property print.timestamp=true ^
--from-beginning
```

Aguarde dois minutos ou mais, e liste novamente os registros utilizando um dos comandos anteriores.

Após um certo tempo, se nenhum outro registro foi produzido, àqueles registros talvez ainda estejam presentes no tópico. Isso acontece porque o Kafka não processa segmentos ativos, que é àquele que está aberto e recebendo os dados que estão chegando. Por esse motivo configuramos `segment.ms`, caso contrário não seria possível visualizar o funcionamento.

Também existe um outro detalhe: o kafka só cria um novo segmento, mesmo que tenha atingido segment.ms, quando novos registros são produzidos. Ele faz isso para economizar processamento, pois não faz sentido criar um novo segmento se não existem dados para serem persistidos.

> A lição que podemos tirar deste experimento é que o tempo de retenção é um valor de referência ou um valor mínimo. Não é algo rigorosamente seguido onde os segmentos deixam de existir exatamente após a expiração do `retention.ms`.

Experimento c

Crie um arquivo chamado `registros.txt`, no diretório atual. Em seguida salve este conteúdo:

```
a:Registro_a_1-fica
b:Registro_b_1
c:Registro_c_1
b:Registro_b_2-fica
c:Registro_c_2-fica
d:Registro_d_1
e:Registro_e_1-fica
d:Registro_d_2-fica
```

Vamos criar o tópico `compactar.n` com a configuração `cleanup.policy=compact`, que ativa a compactação de log. Mas vamos observar que aparentemente ela não estará funcionando.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-compactar.n \
  --config cleanup.policy=compact
```

Windows:


```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-compactar.n ^
--config cleanup.policy=compact
```

- Nenhuma configuração personalizada, somente `cleanup.policy` igual a `compact`

Produza registros no tópico `WB-compactar.n`:

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.n \
  --property "parse.key=true" \
  --property "key.separator=:" < registros.txt
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.n ^
--property "parse.key=true" ^
--property "key.separator=:" < registros.txt
```

Também vamos tentar produzir uma mensagem sem chave (key) no tópico `compactar.n`.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.n
> [Digite qualquer texto e tecle ENTER]
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.n
> [Digite qualquer texto e tecle ENTER]
```

Para tópicos com a compactação de log, qualquer registro sem chave será rejeitado e um erro similar a este devolvido.

```
ERROR Error when sending message to topic compactar.n with key: null, value: 8 bytes with
    error: (org.apache.kafka.clients.producer.internals.ErrorLoggingCallback)
org.apache.kafka.common.InvalidRecordException: This record has failed the validation on
    broker and hence be rejected.
```

Liste os registros para certificar que eles estão lá.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.n \
  --property print.key=true \
  --property print.timestamp=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.n ^
--property print.key=true ^
--property print.timestamp=true ^
--from-beginning
```

Aguarde **mais de dois minutos** e liste os registros novamente. Então será possível observar que os registros ainda não foram compactados.

Isso acontece porque um segmento está ativo. Kafka não compacta segmentos ativos e por essa razão nossa configuração aparentemente não teve efeito.

A compactação acontecerá no momento em que um novo segmento for gerado, algo que acontecerá somente após 7 dias, conforme o valor padrão da configuração `segment.ms` pois ela não foi alterada na criação do tópico. Ou se algum novo registro for produzido.

O que tiramos disso é que nenhuma lógica de negócios deverá ser fortemente dependente do mecanismo de compactação para remover registros antigos em nome de outro, mais recente. Portanto, lembre-se que mesmo após produzir um registro recente com a mesma chave que outros antigos, estes últimos ainda permaneceram algum tempo no log.

Experimento d

Agora vamos criar o tópico `WB-compactar.s`, também com a compactação de log ativada. E desta vez será possível observá-la acontecendo.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-compactar.s \
  --config cleanup.policy=compact \
  --config segment.ms=10000
```

Windows:


```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-compactar.s ^
--config cleanup.policy=compact ^
--config segment.ms=10000
```

- `segment.ms=10000` que é equivalente a 10 segundos, então novos segmentos dentro deste período
- isso permitirá observar a compactação acontecendo

Produza registros no tópico `compactar.s`

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.s \
  --property "parse.key=true" \
  --property "key.separator=:" < registros.txt
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.s ^
--property "parse.key=true" ^
--property "key.separator=:" < registros.txt
```

Liste os registros para certificar que eles estão lá.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.s \
  --property print.key=true \
  --property print.timestamp=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.s ^
--property print.key=true ^
--property print.timestamp=true ^
--from-beginning
```

Observe que mesmo com segmentos gerados a cada 10 segundos, a compactação ainda não aconteceu.

Então aguarde pelo **menos um minuto** e produza um novo registro no tópico.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-compactar.s \
  --property "parse.key=true" \
  --property "key.separator=:"
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-compactar.s ^
--property "parse.key=true" ^
--property "key.separator=:"
```

Exemplo do registro que você deverá digitar:

```
> z:Registro_z_1-fica
```

Então, consuma o tópico `compactar.s` novamente utilizando um dos comandos já citados.

Observe que a compactação foi realizada. Isso acontece porque o kafka também pode criar um novo segmento com base no `segment.ms` somente quando um novo registro é produzido, ou seja, mesmo que o tempo máximo tenha sido atingido nada é feito até que exista necessidade. Desse modo há economia no processamento do servidor.

Experimento e

Crie o tópico `WB-alterar-cfg`, para experimentar a alteração de configurações.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-alterar-cfg
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-alterar-cfg
```

Produza registros.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-alterar-cfg
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-alterar-cfg
```

Consuma os registros produzidos.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-alterar-cfg \
  --property print.key=true \
  --property print.timestamp=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-alterar-cfg ^
--property print.key=true ^
--property print.timestamp=true ^
--from-beginning
```

Altere suas configurações para ativar compactação de log e tempo de vida dos segmentos para 10 segundos.

Linux:

```
kafka-topics.sh --alter \
  --zookeeper z0.kafka.ml:2181 \
  --topic WB-alterar-cfg \
  --config cleanup.policy=compact \
  --config segment.ms=10000
```

Windows:

```
kafka-topics.bat --alter ^
--zookeeper z0.kafka.ml:2181 ^
--topic WB-alterar-cfg ^
--config cleanup.policy=compact ^
--config segment.ms=10000
```

- alteração dos tópicos só é possível através do zookeeper, por este motivo utilizamos o argumento `--zookeeper`

Liste a configuração do tópico, assim poderemos observar que tudo foi alterado e aplicado.

Linux:

```
kafka-topics.sh --describe \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-alterar-cfg
```

Windows:

```
kafka-topics.bat --describe ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-alterar-cfg
```

Produza mais registros. Só que agora a chave será requerida porque nosso tópico está compactado.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-alterar-cfg \
  --property "parse.key=true" \
  --property "key.separator=:"
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-alterar-cfg ^
--property "parse.key=true" ^
--property "key.separator=:"
```

Consuma o tópico novamente com um dos comandos `kafka-console-consumer`. Com isso será possível observar que todos os registros sem chave foram eliminados, assumindo a nova configuração aplicada.

Observar funcionamento

Se você deseja observar o funcionamento, ou seja, **produzir muitos registros** de uma só vez. Um modo simples é fazer uma produção massiva utilizando o comando `kafka-producer-perf-test`.

Exemplo Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-meu.topico \
  --num-records 5000 \
  --record-size 100 \
  --throughput -1 \
  --print-metrics \
  --producer-props acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```


Exemplo Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-meu.topico ^
--num-records 5000 ^
--record-size 100 ^
--throughput -1 ^
--print-metrics ^
--producer-props acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```


Anatomia dos Registros

Dados dentro do kafka são persistidos através dos Registros e agora veremos em detalhes a sua estrutura e a utilidade de seus metadados, cabeçalhos e dados.

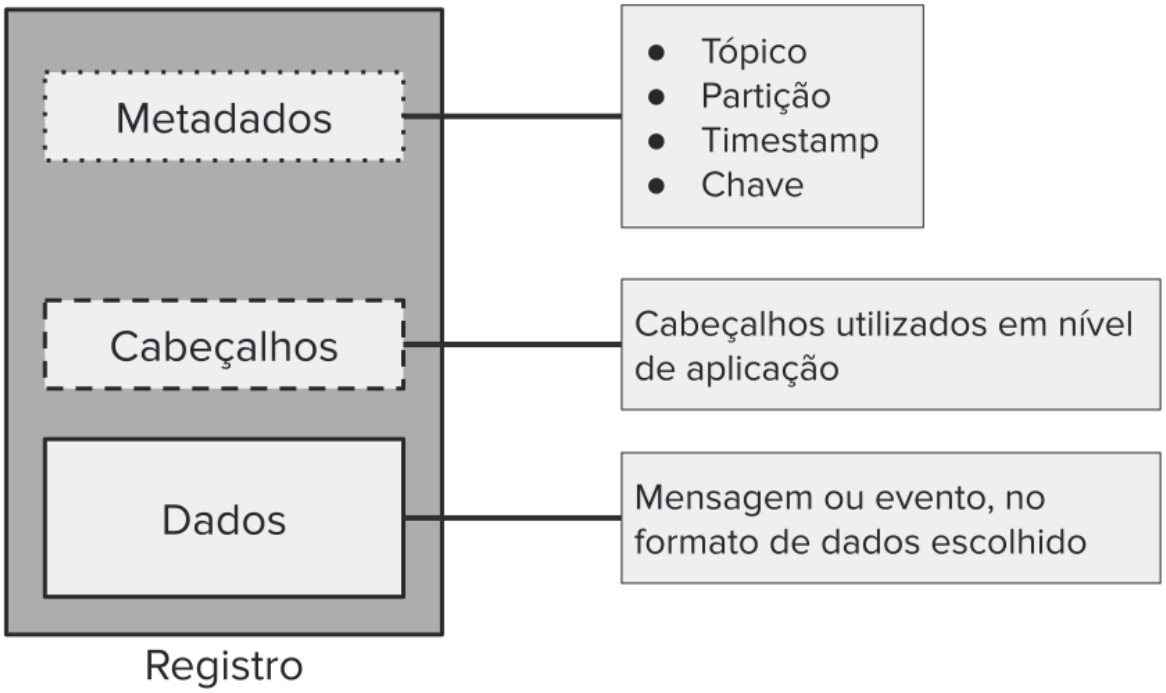

Figura 21: Estrutura de um registro

Um registro no Kafka é composto por **metadados**, **cabeçalhos** e **dados**.

Metadados

São atributos especiais que serão utilizados pelo kafka para operar o registro dentro do cluster. Com eles são definidos qual será o tópico, onde serão persistidos, ou até em qual partição desejamos colocá-los.

tópico - topic

> Na estrutura do registro esse é o metadado **requerido**

O tópico do registro, deve possuir somente letras, maiúsculas ou minúsculas, números, `-`, `.` e `_` com no máximo `255` caracteres. Quaisquer outros caracteres serão rejeitados, provocando erro na criação do tópico.

partição - partition

> Na estrutura do registro esse é o metadado **opcional**

partição dentro do tópico onde o registro está fisicamente.

timestamp

> Na estrutura do registro esse é o metadado **opcional**

Carimbo data-hora quando o registro foi produzido, que pode ser sobrescrito pelo kafka caso a configuração `message.timestamp.type` seja igual a `LogAppendTime`.

chave - key

> Na estrutura do registro esse é o metadado **opcional**

A chave não é o identificador único do registro (dados), ela é um valor utilizado no algoritmo responsável pela distribuição dos registros nas partições e pela compactação de log. Ela pode ser um texto, um número ou qualquer tipo que nosso caso de uso venha a necessitar.

Kafka garante, na configuração padrão, que todos os registros com a mesma chave sempre serão persistidos na mesma partição.

- para determinar a partição com base na chave, o Kafka aplica uma função hash, onde a entrada é o valor da chave e a saída o número da partição.

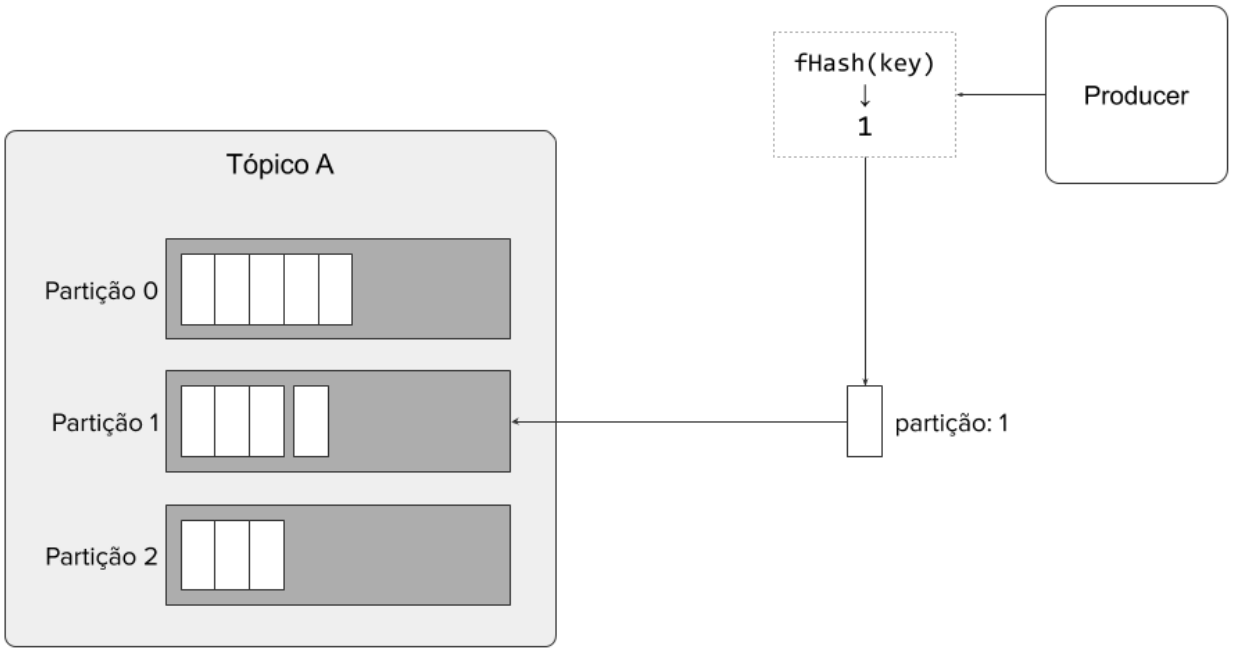

Figura 22: Determinando a partição com base na chave

Cabeçalhos

Cabeçalhos são criados e processados em nível de aplicação, ou seja, você define quais são seus nomes, tipos e como usá-los. O kafka somente se encarrega de persisti-los.

> São arranjos (array) de bytes para o kafka, não existindo qualquer tratamento, roteamento ou transformação em seus valores.

Cabeçalhos são definidos por um nome e um valor, exemplos:

Nome	Valor
id	1ab8bb8f87f7
fonte	/compras
tipo	compras.registrada
numero	3

Não há restrições técnicas. Crie quantos forem necessários e sempre utilize-os para enriquecer sua semântica de comunicação.

> Para eventos, existe uma spec aberta chamada CloudEvents.

Dados

Área de dados, onde segue aquilo que você definiu como formato, estrutura e que transmitem a semântica escolhida. Também trata-se de um conjunto de bytes, que em nenhum momento é processado pelo kafka, só persistido. Esses dados tem significado único e exclusivo para as aplicações produtoras e consumidoras.

Pode-se utilizar qualquer formato de dados, encarregando *producers* e *consumers* da serialização e deserialização, respectivamente. Veremos detalhes nas seções Produtores e Consumidores.

Os formatos de dados mais utilizados são AVRO e JSON. Ambos possuem prós e contras, vale sempre olhar para o caso de uso e entender qual deles resolverá melhor o problema.

Exemplo de dados no formato JSON:

```
{
    "cliente-id" : "C9b0f",
    "valor"      : 80.0000,
    "compra-id"  : "4c40"
}
```

Quando o exemplo acima for definido como dados de um registro, quem o produzir deverá se responsabilizar por transformá-lo em um conjunto de bytes. Desse modo o que chegará no broker serão somente bytes, ou seja, o kafka não "entenderá" o formato de dados empregado no registro.

Figura 23: Exemplo de um registro

No exemplo acima temos um registro kafka, onde utilizamos o metadado tópico, cabeçalhos e uma área de dados com um formato json.

Produtores - *Producer*

Produtores, publicadores, criadores, ou *producer* em inglês, são as fontes de dados de onde originam-se os dados presentes nos tópicos. São as aplicações que criam dados e os enviam para o kafka, são pipelines de dados em uma arquitetura DataOps e são os serviços em um ecossistema de microsserviços.

Via de regra, qualquer software que envie uma sequência de bytes correta para kafka já é um producer, mas fazer isso é trabalhoso e certamente consumiria muitas horas de projeto. Por este motivo existe a Producer API, uma definição sobre como um *producer* deve funcionar e está disponível em várias linguagens.

Serialização

Serialização é uma operação que recebe uma estrutura, como uma classe, um json, um avro, um texto, etc e os transforma em um arranjo (array) de bytes dispostos em série.

Isso é necessário para levar até o kafka somente os bytes, assim como vimos na Anatomia dos Registros.

Principais configurações

Já utilizamos algumas configurações importantes, sem entrar em detalhes. Agora abordaremos como cada uma delas, contribui para o setup correto no momento de produzir registros no Kafka.

key.serializer

Nome da classe que será utilizada para serializar a chave de cada registro.

Essa é uma configuração muito utilizada quando a aplicação produtora foi escrita em uma linguagem fortemente tipada, como: Java e C#. E é comum já existirem implementações para tipos como texto (string), número inteiro (integer, long) e números fracionários (float, double).

- valor padrão: *não há*
  - **requerido**: sempre devemos definir seu valor
- implementações mais usadas e disponíveis no Java
  - Texto: `org.apache.kafka.common.serialization.StringSerializer`
  - Número: `org.apache.kafka.common.serialization.IntegerSerializer`
  - Número (longo): `org.apache.kafka.common.serialization.LongSerializer`
  - Fracionários: `org.apache.kafka.common.serialization.FloatSerializer`
  - Fracionários (longo): `org.apache.kafka.common.serialization.DoubleSerializer`

Exemplo:

```
key.serializer=org.apache.kafka.common.serialization.StringSerializer
```

Em outras linguagens, como C#, não é necessário definir explicitamente essa configuração para tipos comuns, como àqueles que já citamos. Somente temos que nos preocupar caso seja utilizado alguma classe ou estrutura complexa, algo pouco comum para definição de chaves no Kafka.

> Mesmo que formos produzir registros sem chave, é necessário definirmos um valor para essa configuração, sempre.

value.serializer

Nome da classe utilizada para serialização dos valores presentes em todos os registros produzidos no kafka.

Assim como a configuração `key.serializer`, também é necessário definir qual o serializador para o valor. Podemos utilizar as mesmas classes, mas aqui é muito comum existir uma estrutura complexa para representação dos dados de negócio. Para isso devemos escolher um formato de dados:

- os tipos complexos mais comuns são JSON ou AVRO.

> Veremos exemplos com JSON e AVRO nos laboratórios.

- valor padrão: *não há*
  - **requerido**: sempre devemos definir seu valor
- implementações mais usadas e disponíveis no Java
  - Texto: `org.apache.kafka.common.serialization.StringSerializer`
  - Número: `org.apache.kafka.common.serialization.IntegerSerializer`
  - Número (longo): `org.apache.kafka.common.serialization.LongSerializer`
  - Fracionários: `org.apache.kafka.common.serialization.FloatSerializer`
  - Fracionários (longo): `org.apache.kafka.common.serialization.DoubleSerializer`

Exemplo:

```
value.serializer=org.apache.kafka.common.serialization.StringSerializer
```

Em outras linguagens, como C#, não é necessário definir explicitamente essa configuração para tipos comuns, como àqueles que já citamos. Somente temos que nos preocupar caso seja utilizado alguma classe ou estrutura complexa, algo pouco comum para definição de chaves no Kafka.

acks

O número de conhecimentos, ou *acknowledgement* em inglês, sobre o recebimento dos registros pelos brokers do cluster. Isso significa que ao enviar um *ack* o broker garante que os registros foram persistidos e estamos seguros de que não serão perdidos.

- valor padrão: `1`
- valores possíveis
  - `0`: nenhum conhecimento será aguardado logo após enviar os registros
  - `1`: será aguardado pelo menos um conhecimento
  - `all`: serão aguardados o conhecimento de todos os brokers do cluster
- configurações relacionadas
  - `delivery.timeout.ms`
  - `min.insync.replicas`, uma configuração do tópico

Cada valor desta configuração tem um impacto na produção de registros:

- `acks=0`, são para casos de uso onde a perda de dados é aceitável, como métricas e logs de aplicações, pois a taxa de transferência será altíssima
- `acks=1`, são para casos de uso onde é necessária confiabilidade, mas ainda sim manter uma boa taxa de transferência. Com este valor ainda existe a possibilidade de perder dados, mesmo que baixa, porque o broker que respondeu ack poderá falhar antes da replicação para seus pares.
- `acks=all`, são para situações onde não existe tolerância para perda de dados. Como em um ecossistema empresarial onde são realizadas transações de negócio através de microsserviços.

bootstrap.servers

Lista com nome de host, ou ip, e porta, para brokers kafka separados por vírgula: ,

Essa configuração é utilizada para a descoberta do cluster, por este motivo sempre é melhor fornecer ao menos dois hosts. Assim não teremos problemas caso um deles esteja parado, por exemplo.

- valor padrão: *não há*
  - **requerido**: sempre devemos definir seu valor

Exemplo:

```
bootstrap.servers=127.0.0.1:9092,kafka2.ijo:9094
```


buffer.memory

Tamanho máximo em bytes que será utilizado para guardar registros antes de serem enviados ao kafka.

- valor padrão: `33554432 (~34MB)`
- configurações relacionadas
  - `max.block.ms`

compression.type

Tipo de compressão para lotes de registros. Com essa configuração é possível diminuir o tamanho dos dados antes de serem enviados ao broker.

Isso ajuda a otimizar o consumo da rede e, dependendo da configuração no tópico, também otimiza o tamanho ocupado na persistência física nos segmentos.

- valor padrão: `none`
- valores possíveis
  - `none`: nenhuma compressão é aplicada, mantendo o tamanho original
  - `gzip`: ótima compressão, demandando mais processamento
  - `snappy`: compressão criada pelo Google
  - `lz4`: boa relação entre tamanho e processamento
  - `zstd`: compressão criada pelo Facebook

A compressão quando aplicada registro-a-registro, que significa desligar o loteamento (`batch.size=0`), afeta drasticamente a taxa de transferência e a eficiência do padrão utilizado, *compression ratio* em inglês.

retries

Número máximo de retentativas para enviar o lote de registro que falhou. Sempre que ao tentar produzir dados no kafka, houverem erros que possam ser recuperados, como uma instabilidade na rede, acontecerá automaticamente a retentativa.

> Erros não recuperáveis falham imediatamente.

- valor padrão: `2147483647`
- configurações relacionadas
  - `request.timeout.ms`
  - `max.in.flight.requests.per.connection`

batch.size

Limite superior máximo para o tamanho em bytes que os lotes podem atingir antes de serem enviados. Lotes são montados por partição, então este é o limite do que será enviado a uma partição, ou seja, podem existir vários lotes com esse tamanho máximo.

- valor padrão: `16384 (~16KB)`
  - configure `0` para desligá-la e não lotear os registros
- configurações relacionadas
  - `linger.ms`
  - `max.message.bytes`, uma configuração do tópico

Ao desligar o loteamento de registros a taxa de transferência é drasticamente afetada, bem como a eficiência na compressão de dados, caso ele seja utilizada.

client.id

Identificador do producer, que é puramente informativo para logs e métricas. Nenhuma validação ou gerenciamento é feito com base nesta configuração.

- valor padrão: *não há*

Utilize um valor com significado. Exemplos bons são:

- `clientes.bff`
- `clientes.svc`

Exemplos ruins são:

- `app`
- `serviço`
- `6e47f11e-69a8-4389-acc0-7f4fbd14c454`

linger.ms

Tempo em milisegundos para aguardar a formação de lotes, evitando-se assim a produção de requisições menores em situações onde a carga de trabalho está moderada ou baixa. Se lê esta configuração como um atraso para enviar o lote corrente ao Kafka.

- valor padrão: `0`
- configurações relacionadas
  - `batch.size`

max.request.size

Define o tamanho máximo em bytes para uma requisição. Ela configura um limite máximo de lotes de registros que podem ser enviados em apenas uma requisição.

- valor padrão: `1048576` (~`1MB`)
- configurações relacionadas
  - `batch.size`
- issues relacionadas
  - 8350 - 06/ABR/2019: lotes maiores onde o valor definido no broker causam erro, provocando tentativas deliberadas para a divisão sem respeitar esta configuração. Assim, o erro RecordTooLargeException só acontece quando um único registro é maior que o valor definido.

request.timeout.ms

Tempo máximo em milissegundos para aguardar uma resposta da requisição, então senão chegar a ela durante este tempo haverá retentativas ou falha se já terminaram.

- valor padrão: `30000` (30 segundos)
- configurações relacionadas
  - `retries`
  - `delivery.timeout.ms`
  - `replica.lag.time.max.ms` (uma configuração no broker)

delivery.timeout.ms

Tempo máximo em milissegundos para enviar os dados, aguarda retorno do conhecimento de recebimento, o *ack* quando ele foi configurado e o tempo para retentativas de envio para erros recuperáveis.

- valor padrão: `120000` (2 minutos)
- configurações relacionadas
  - `acks`=1, `acks=all`
  - `request.timeout.ms`
  - `linger.ms`

Seu valor deve ser maior que a soma das configurações `linger.ms` e `request.timeout.ms`.

enable.idempotence

Quando habilitada haverá a garantia de que os lotes serão produzidos exatamente uma vez, caso contrário haverão retentativas que poderão modificar a ordenação dos registros.

- valor padrão: `false`
- valores possíveis
  - `true`
  - `false`
- configurações relacionadas quando `true`
  - `acks=all`
  - `retries>0`
  - `max.in.flight.requests.per.connection<=5`

Ao configurar `true` se as outras configurações não foram definidas como o descrito acima, será lançada uma exceção de execução e não será possível produzir dados no kafka. Porém, todas as outras serão automaticamente configuradas com valores consistentes para este cenário de uso, desde que nenhum outro valor tenha sido informado.

max.in.flight.requests.per.connection

Número máximo de requisições sem conhecimento, o *ack*, que podem ser enviadas na mesma conexão, ou seja, estão em voo e aguardando finalização. Quando este valor máximo for atingido, novos envios ficarão bloqueados até que os atuais sejam resolvidos com a chegada do conhecimento ou com uma falha não recuperável.

- valor padrão: `5`
- configurações relacionadas
  - `enable.idempotence`
  - `retries`

Quando utilizamos valores superiores à `0` e também retentativas, `retries > 0`, a garantia de ordem não se mantém. Imagine o cenário:

- lote 1 p/ partição A é enviado e fica em voo até o ack
- lote 2 p/ partição A é enviado e fica em voo
- lote 1 falha e segue para retentativa
- lote 2 é bem sucedido
- neste instante a garantia de ordem for perdida, porque registros produzidos depois foram armazenados antes do primeiro lote
- por este motivo temos que entender se o caso de uso tolera a não-garantia da ordenação

Lab 3: produzindo registros

Já produzimos e consumimos muitos registros até agora. Neste laboratório vamos entender outros detalhes sobre sua estrutura.

Experimento a

Crie o tópico `WB-distribuir` para testes sobre a distribuição dos registro sem chave.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 4 \
  --topic WB-distribuir
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 4 ^
--topic WB-distribuir
```

Produza alguns registros sem chave. No exemplo abaixo utilizamos o `kafka-producer-perf-test`, mas você poderá perfeitamente empregar o `kafka-console-producer`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-distribuir \
  --num-records 5 \
  --record-size 100 \
  --throughput -1 \
  --print-metrics \
  --producer-props acks=1 \
      batch.size=0 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-distribuir ^
--num-records 5 ^
--record-size 100 ^
--throughput -1 ^
--print-metrics ^
--producer-props acks=1 ^
batch.size=0 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```


Experimento b

Crie o tópico `WB-chaves` para experimentar como registros com a mesma chave seguem para a mesma partição.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 4 \
  --topic WB-chaves
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 4 ^
--topic WB-chaves
```

Então produza eventos com chave de partição. Lembrando de repetir algumas delas para observar o funcionamento.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-chaves \
  --property "parse.key=true" \
  --property "key.separator=:"
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-chaves ^
--property "parse.key=true" ^
--property "key.separator=:"
```

Visualise como foi a distribuição dos eventos com base em suas chaves de partição.

Experimento c

Crie o tópico `WB-chaves-cfg` para experimentar como afetará a distribuição dos registros com a mesma chave ao alterar um tópico e adicionar novas partições.

Linux:


```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 4 \
  --topic WB-chaves-cfg
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 4 ^
--topic WB-chaves-cfg
```

Produza eventos com chave de partição, lembrando de repetir algumas delas para observar o funcionamento.

Linux:

```
kafka-console-producer.sh \
  --broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-chaves-cfg \
  --property "parse.key=true" \
  --property "key.separator=:"
```

Windows:

```
kafka-console-producer.bat ^
--broker-list b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-chaves-cfg ^
--property "parse.key=true" ^
--property "key.separator=:"
```

Visualise como foi a distribuição dos eventos com base em suas chaves de partição.

Altere o número de partições para `5`, ou seja, crie uma nova partição no tópico `WB-chaves-cfg`:

Linux:

```
kafka-topics.sh --alter \
  --zookeeper z0.kafka.ml:2181 \
  --partitions 5 \
  --topic WB-chaves-cfg
```

Windows:

```
kafka-topics.bat --alter ^
--zookeeper z0.kafka.ml:2181 ^
--partitions 5 ^
--topic WB-chaves-cfg
```

Produza novos eventos com as mesmas chaves utilizadas antes de modificar o número de partições e visualize novamente como foi a distribuição dos eventos.

Experimento d

Entendendo o impacto da configuração acks na taxa de transferência.

Tópico `WB-acks-0` para experimentar `acks=0`.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 8 \
  --topic WB-acks-0
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 8 ^
--topic WB-acks-0
```

Produzir cinco mil registros com kafka-producer-perf-test, cada um deles com `10KB`, utilizando valores padrão para todas as outras configurações do producer.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-acks-0 \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props acks=0 \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-acks-0 ^
--num-records 5000 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props acks=0 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Nosso resultado será impresso como o exemplo abaixo. Isso varia de acordo com seu poder de processamento, bem como a velocidade da sua rede.

```
5000 records sent, 1657.824934 records/sec (16.19 MB/sec), 843.45 ms avg latency, 1414.00 ms
    max latency, 859 ms 50th, 1284 ms 95th, 1340 ms 99th, 1411 ms 99.9th.
```

Tópico `WB-acks-1` para experimentar `acks=1`.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 8 \
  --topic WB-acks-1
```

Windows:

```
kafka-topics.bat --create ^
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
  --replication-factor 3 ^
  --partitions 8 ^
  --topic WB-acks-1
```

Produzir cinco mil registros com kafka-producer-perf-test, cada um deles com `10KB`, utilizando valores padrão para toda as outras configurações do producer.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-acks-1 \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props acks=1 \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
  --topic WB-acks-1 ^
  --num-records 5000 ^
  --record-size 10240 ^
  --throughput -1 ^
  --producer-props acks=1 ^
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Nosso resultado será impresso como o exemplo abaixo. Isso varia de acordo com seu poder de processamento, bem como a velocidade da sua rede.

```
5000 records sent, 1381.978994 records/sec (13.50 MB/sec), 1073.32 ms avg latency, 1608.00
    ms max latency, 1134 ms 50th, 1468 ms 95th, 1543 ms 99th, 1589 ms 99.9th.
```

Tópico `WB-acks-all` para experimentar `acks=all`.

Linux:


```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 8 \
  --topic WB-acks-all
```

Windows:

```
kafka-topics.bat --create ^
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
  --replication-factor 3 ^
  --partitions 8 ^
  --topic WB-acks-all
```

Produzir cinco mil registros com `kafka-producer-perf-test`, cada um deles com `10KB`, utilizando valores padrão para todas as outras configurações do producer.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-acks-all \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props acks=all \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
  --topic WB-acks-all ^
  --num-records 5000 ^
  --record-size 10240 ^
  --throughput -1 ^
  --producer-props acks=all ^
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Nosso resultado será impresso como o exemplo abaixo. Isso varia de acordo com seu poder de processamento, bem como a velocidade da sua rede.

```
5000 records sent, 1090.512541 records/sec (10.65 MB/sec), 1368.86 ms avg latency, 2112.00
    ms max latency, 1467 ms 50th, 1933 ms 95th, 2053 ms 99th, 2103 ms 99.9th.
```

Uma observação interessante é a tendência que existe, ou seja, quanto maior o número de `acks` necessário, menor será a taxa de transferência.

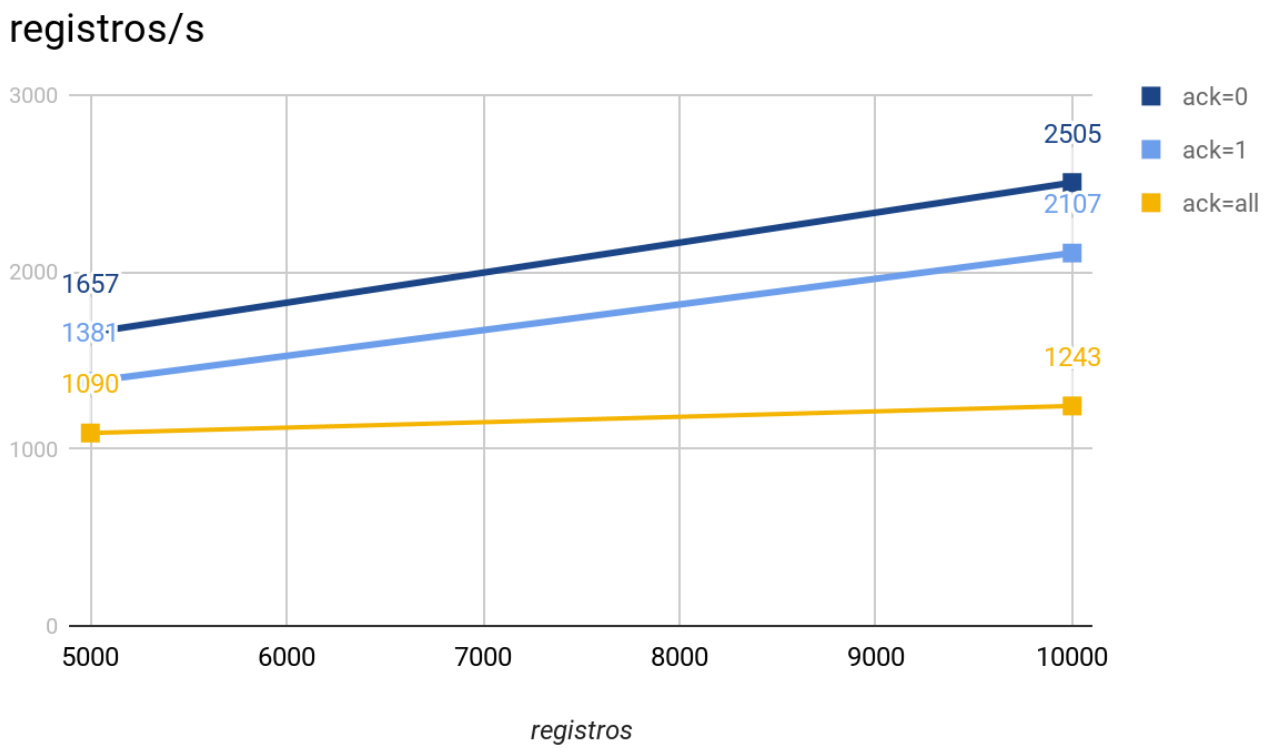

Figura 24: Taxa de transferência por tipo de acks

Experimento e

Neste experimento vamos entender o mecanismo de compressão e qual sua relação com a mesma configuração presente no tópico.

Crie o tópico `WB-compressao-producer` que assume a compressão, se ela foi empregada, criada pela aplicação produtora.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-compressao-producer \
  --config compression.type=producer
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-compressao-producer ^
--config compression.type=producer
```


Liste o conteúdo físico deste tópico, assim verificaremos que o arquivo de log ainda está vazio e teremos uma base de comparação após produzir os registros.

Linux:

```
ls -alh /tmp/broker1-logs/WB-compressao-producer-0
```

Windows:

```
dir c:\temp\broker1-logs\WB-compressao-producer-0
```

Produzir registros no tópico `WB-compressao-producer`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-compressao-producer \
  --num-records 300 \
  --record-size 102400 \
  --throughput -1 \
  --print-metrics \
  --producer-props \
      acks=all \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      compression.type=gzip \
      batch.size=1024000
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-compressao-producer ^
--num-records 300 ^
--record-size 102400 ^
--throughput -1 ^
--print-metrics ^
--producer-props ^
acks=all ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
compression.type=gzip ^
batch.size=1024000
```

- `102400` é o equivalente a `100KB`
- `1024000` é o equivalente a `10 * 100KB`
- `300 * 100KB` será a quantidade de bytes produzidas antes da compressão, algo em torno de 30MB
- `gzip` entrega bons níveis de compressão, usando um pouco mais de processamento

Liste novamente o diretório onde estão os arquivos de log. Será possível visualizar que o espaço ocupado pelo log é bem inferior aos `30MB` produzidos, graças a compressão gzip que foi realizada no *producer* e mantida pelo Kafka. Neste experimento economizamos o uso da rede, bem como espaço no disco dos servidores.

Agora, crie o tópico `WB-uncompressed`, que está configurado para não comprimir os registros ao persisti-los fisicamente nos arquivos de log. Isso significa que dados que foram comprimidos pelo producer serão descomprimidos ao chegarem no kafka.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-uncompressed \
  --config compression.type=uncompressed
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-uncompressed ^
--config compression.type=uncompressed
```

Produzir registros no tópico `WB-uncompressed`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-uncompressed \
  --num-records 300 \
  --record-size 102400 \
  --throughput -1 \
  --print-metrics \
  --producer-props \
      acks=all \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      compression.type=gzip \
      batch.size=1024000
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-uncompressed ^
--num-records 300 ^
--record-size 102400 ^
--throughput -1 ^
--producer-props ^
acks=all ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
compression.type=gzip ^
batch.size=1024000
```

Então, liste o conteúdo físico deste tópico, assim verificaremos qual o tamanho do log.

> Caso o arquivo ainda seja listado com zero bytes, aguarde alguns segundos. Isso é necessário porque a descarga da memória ainda não foi para o arquivo.

Linux:

```
ls -alh /tmp/broker1-logs/WB-uncompressed-0
```

Windows:

```
dir c:\temp\broker1-logs\WB-uncompressed-0
```

Fica claro que a economia foi somente no uso da rede, porque a configuração `compression.type=uncompressed` do tópico fez com que o lote fosse descomprimido e armazenado em seu tamanho original, ocupando `~30MB`.

Experimento f

Visualizar a eficiência quando utilizamos loteamento de registros através da configuração `batch.size` do *producer*.

Crie o tópico `WB-lote100kb`, que possui o limite máximo de `100KB` para lotes de registros em cada partição.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 1 \
  --topic WB-lote100kb \
  --config max.message.bytes=102400
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 1 ^
--topic WB-lote100kb ^
--config max.message.bytes=102400
```

- `102400` é o equivalente a `100KB`

Produzir `5000` registros com lote de tamanho máximo igual a `90KB`.

Linux:


```
kafka-producer-perf-test.sh \
  --topic WB-lote100kb \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      batch.size=92160
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-lote100kb ^
--num-records 5000 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
batch.size=92160
```

Nosso resultado será impresso como o exemplo abaixo. Isso varia de acordo com o seu poder de processamento, bem como a velocidade da sua rede.

```
5000 records sent, 2623.294858 records/sec (25.62 MB/sec), 667.88 ms avg latency, 1010.00 ms
    max latency, 789 ms 50th, 997 ms 95th, 1004 ms 99th, 1007 ms 99.9th.
```

Produzir `5000` registros com lote de tamanho máximo igual a `45KB`, metade do primeiro experimento.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-lote100kb \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      batch.size=46080
```

Windows:


```
kafka-producer-perf-test.bat ^
--topic WB-lote100kb ^
--num-records 5000 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
batch.size=46080
```

Resultados, que variam em função do poder de processamento e velocidade da rede:

```
5000 records sent, 1820.167455 records/sec (17.78 MB/sec), 1081.16 ms avg latency, 1636.00
    ms max latency, 1185 ms 50th, 1578 ms 95th, 1626 ms 99th, 1632 ms 99.9th.
```

Produzir `5000` registros sem loteamento de registros, ou seja, `batch.size=0`

Linux:

```
kafka-producer-perf-test.sh \
  --topic lote100kb \
  --num-records 5000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      batch.size=0
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic lote100kb ^
--num-records 5000 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
batch.size=0
```

Resultados, que variam em função do poder de processamento e velocidade da rede:

```
5000 records sent, 629.564341 records/sec (6.15 MB/sec), 3497.45 ms avg latency, 5208.00 ms
    max latency, 4198 ms 50th, 5041 ms 95th, 5182 ms 99th, 5207 ms 99.9th.
```

A tendência é que quanto menor for o tamanho do lote, menor será a taxa de transferência:

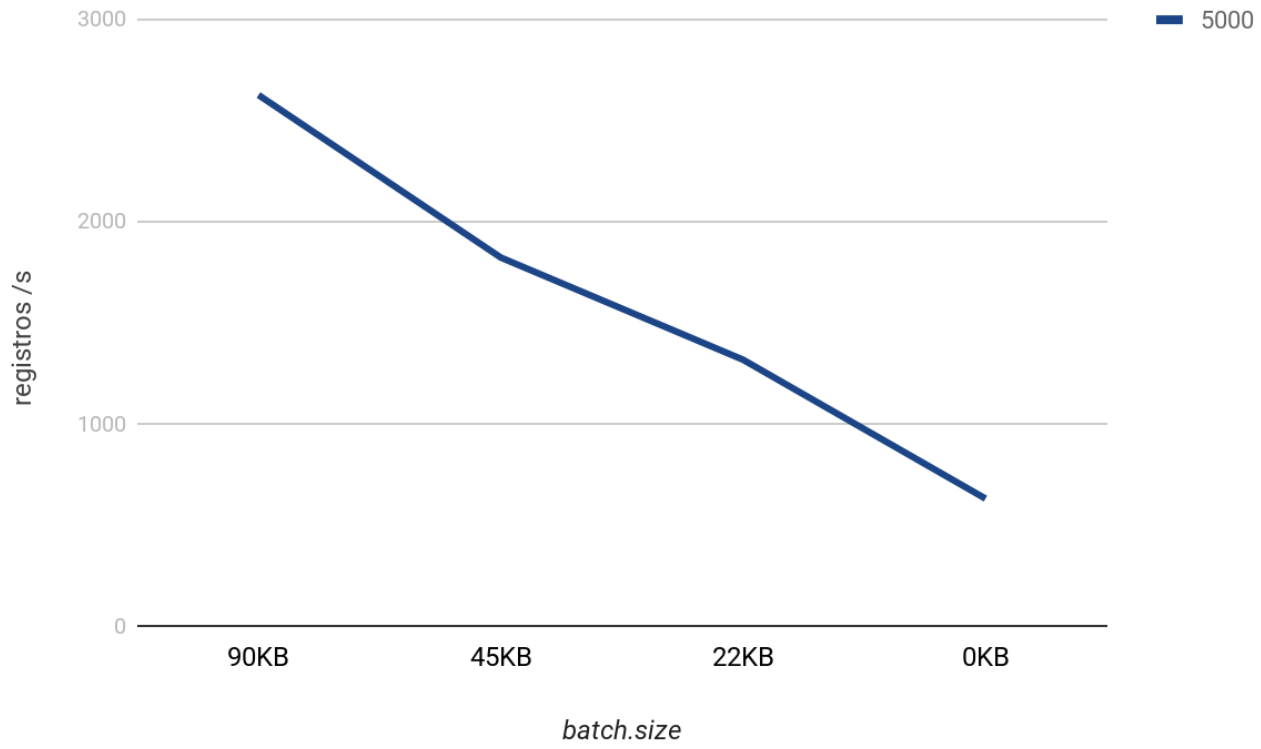

Figura 25: Taxa de transferência vs. tamanho do batch

Experimento g

Entender como a configuração `enable.idempotence` pode impactar na taxa de transferência.

Criar o tópico `WB-idemp` para o experimento.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 7 \
  --topic WB-idemp
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 7 ^
--topic WB-idemp
```

Produzir registros com idempotência ativa.

Linux:


```
kafka-producer-perf-test.sh \
  --topic WB-idemp \
  --num-records 10000 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      enable.idempotence=true
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-idemp ^
--num-records 10000 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
enable.idempotence=true
```

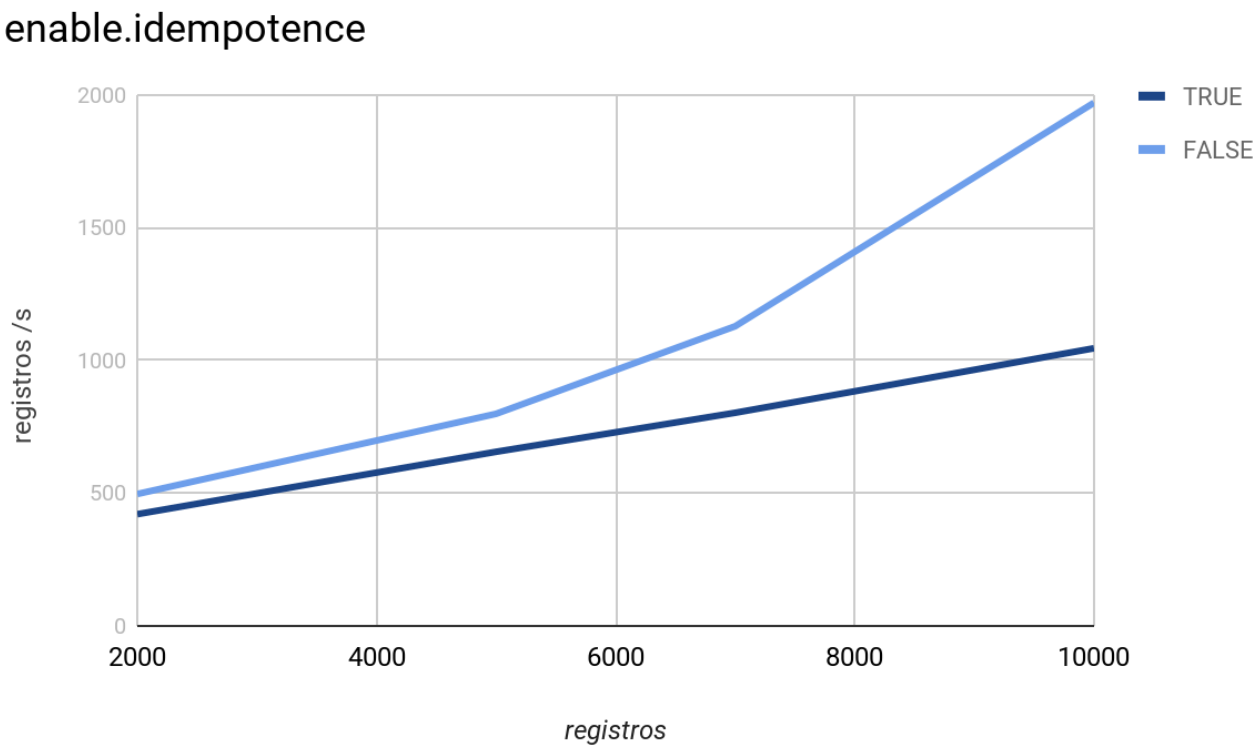

Figura 26: Taxa de transferência vs. idempotência

Experimento h

Experimentar o impacto da idempotência nas configurações do producer.

Definir um valor incorreto para `max.in.flight.requests.per.connection`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-idemp \
  --num-records 100 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
      enable.idempotence=true \
      max.in.flight.requests.per.connection=6
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-idemp ^
--num-records 100 ^
--record-size 10240
--throughput -1 ^
--producer-props ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
enable.idempotence=true ^
max.in.flight.requests.per.connection=6
```

Teremos um erro similar a este:

```
org.apache.kafka.common.config.ConfigException: Must set
    max.in.flight.requests.per.connection to at most 5 to use the idempotent producer.
```

Definir um valor inconsistente para a configuração `acks`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-idemp \
  --num-records 100 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
    enable.idempotence=true \
    max.in.flight.requests.per.connection=5 \
    acks=0
```

Windows:


```
kafka-producer-perf-test.bat ^
--topic WB-idemp ^
--num-records 100 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
enable.idempotence=true ^
max.in.flight.requests.per.connection=5 ^
acks=0
```

Teremos um erro similar a este:

```
org.apache.kafka.common.config.ConfigException: Must set acks to all in order to use the
    idempotent producer. Otherwise we cannot guarantee idempotence.
```

Definir um valor inconsistente para a configuração retries.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-idemp \
  --num-records 100 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
    bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 \
    enable.idempotence=true \
    max.in.flight.requests.per.connection=5 \
    acks=all \
    retries=0
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-idemp ^
--num-records 100 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
enable.idempotence=true ^
max.in.flight.requests.per.connection=5 ^
acks=all ^
retries=0
```

Teremos um erro similar a este:

```
org.apache.kafka.common.config.ConfigException: Must set retries to non-zero when using the
    idempotent producer.
```


Ordenação no kafka

Kafka garante a ordenação em nível de partição e uma mesma origem, ou seja, não existe ordenação global em um tópico. E também estamos tratando de um sistema distribuído, logo, não existe garantia quando existirem vários produtores no mesmo tópico e partição. Isso quer dizer que prevalece a ordem daquele que chegar primeiro.

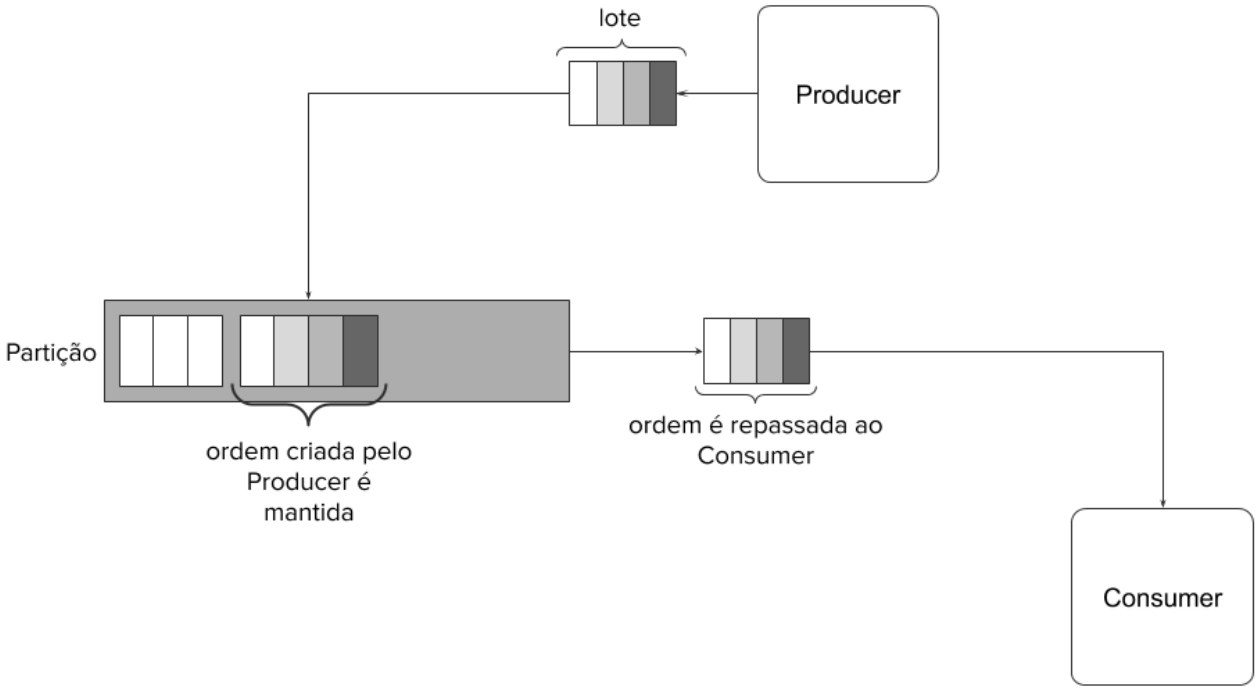

Figura 27: Manutenção da ordem produzida

Lab 4: entendendo a produção de dados com uma App Java

A partir deste lab iniciaremos experimentos com aplicações, mas isso não requer que você seja uma pessoa experiente em programação. Siga as orientações e será possível executar tudo sem problemas.

Vamos aos passos:

- Faça o download do programa Java
  - Produtor Java
  - Extraia o conteúdo no diretório `kafka-m1`

Inicie um console consumer no tópico `WB-lab4-producer` e como ele não existe, será automaticamente criado.

Linux:


```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-lab4-producer \
  --property print.key=true \
  --property print.timestamp=true \
  --property print.value=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-lab4-producer  ^
--property print.key=true ^
--property print.timestamp=true ^
--property print.value=true ^
--from-beginning
```

Agora, execute algumas vezes a aplicação Java.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dproduzir='WB-lab4-producer'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dproduzir='WB-lab4-producer'
```

Volte ao console onde você iniciou o consumer e verifique se os registros foram listados. Caso a saída tenha sido algo similar ao listado abaixo, tudo está correto e a aplicação funcionou.

```
CreateTime:1590416139482    null    Dados do evento
CreateTime:1590416824517    null    Dados do evento
CreateTime:1590416852094    null    Dados do evento
```


Consumidores - Consumer

Consumidores, subscritores, ou *consumers* em inglês, são os destinos dos dados persistidos no Kafka. Estes destinos tem a responsabilidade de consumir lotes de registros e processá-los, gerando resultados ou até mesmo produzindo registros em outros tópicos no Kafka mesmo.

Assim como os produtores, Kafka tem uma Consumer API para que não seja preciso se preocupar em interpretar o protocolo de consumo. Disponível para Java e em várias outras linguagens.

Durante todo o tempo em que um consumidor está conectado ao Kafka ele deve enviar sinais de vida, heartbeats em inglês. Esses sinais de vida informam ao broker que o consumidor ainda está ativo e processando registros, caso contrário os tópicos e partições que ele estava sobrescrito serão atribuídos a outros consumidores. A isso dá-se o nome rebalanceamento de consumo.

Deserialização

Como já é conhecido, o valor e cabeçalhos dos registros são apenas arranjos (array) de bytes. Assim, é necessário transformá-los em uma estrutura que seja amigável para manipulação em nossa linguagem de programação. Deserialização é isso: tomar um arranjo de bytes e transformá-lo em um texto, uma classe, um avro, etc.

É importante saber que a deserialização é ditada pelo formato de dados utilizado na serialização empregada pelo Produtor, ou seja, se produtor utilizou um formato JSON, ele deve ser respeitado pela deserialização.

Grupos de consumo

Uma característica espetacular do Kafka são os grupos de consumo, pois, aplicações mesmo que geograficamente separadas podem participar do mesmo grupo. E existem características importantes que os fazem ser tão simples e ao mesmo tempo robustos.

- todo consumidor sempre deverá informar seu grupo de consumo
- o número total de consumidores em um grupo está limitado a quantidade de partições presentes nos tópicos subscritos
- o mesmo evento jamais será enviado para dois consumidores do mesmo grupo
- o gerenciamento do *offset* é realizado para um grupo de consumo
- é um identificador que informamos através da configuração `group.id`
- sempre que um consumidor entra ou sai do grupo, ele provoca o rebalanceamento

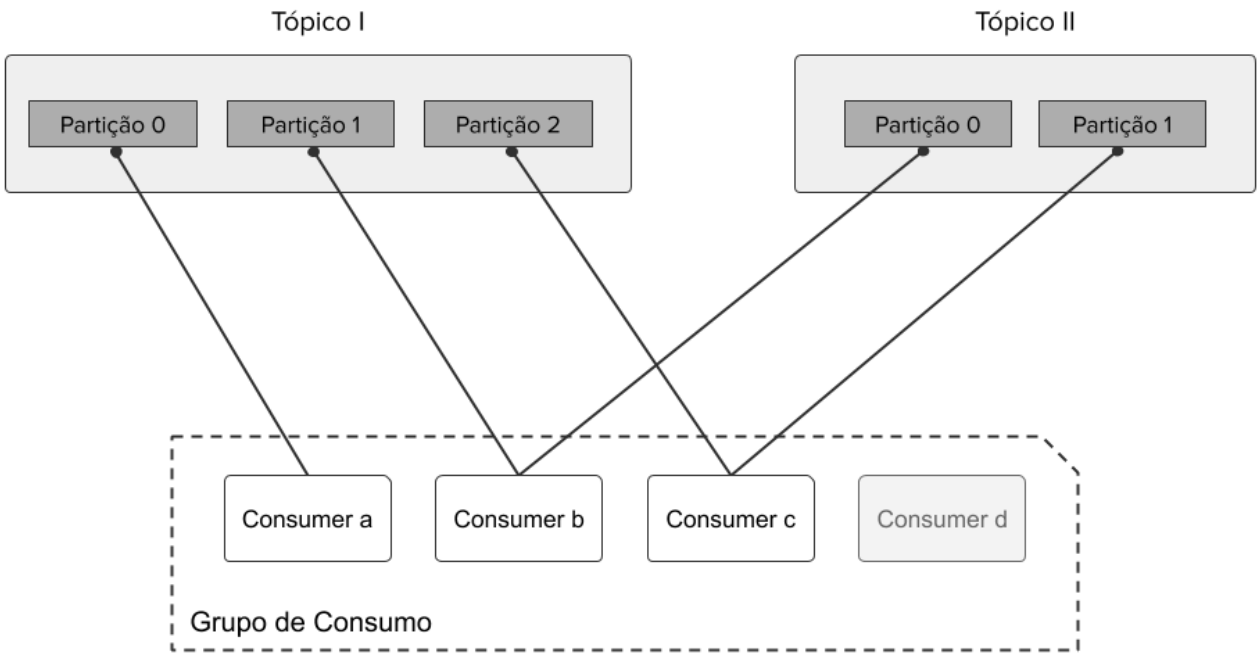

Figura 28: Grupo de Consumo

Rebalanceamento

Balancear o consumo no kafka é uma das suas principais características, pois isso permite paralelizar o processamento dos registros presentes nos tópicos. Sabendo das características que os grupos de consumo têm, estas são as situações que provocam o rebalanceamento de consumo em um grupo:

- quando um consumidor entra no grupo
- quando um consumidor não envia sinal de vida
- quando um consumidor sai do grupo
- quando um consumidor subscreve usando regex e um novo tópico é criado
- quando um consumidor passa muito tempo sem realizar poll

Estratégias de commit

Chama-se *commit* a confirmação processada do maior offset. Ele é análogo ao *ack* recebido pelos produtores. Nesse caso o consumidor está informando ao gerenciamento do offset presente no broker kafka que o lote consumido já foi processado para àquele grupo de consumo ao qual ele pertence, assim, após um rebalanceamento ou failover, os novos consumidores iniciaram o consumo a partir daquele offset.

Existem algumas formas de executar o commit, cada uma delas atendendo casos de uso diferentes, pois a forma escolhida impacta na taxa de transferência, por exemplo. No caso do consumidor, o commit dita o quão rápido ele processa eventos produzidos, ou seja a capacidade de processar duplicatas de uma maneira idempotente.

Veremos três estratégias, as demais requerem abordagens mais avançadas.

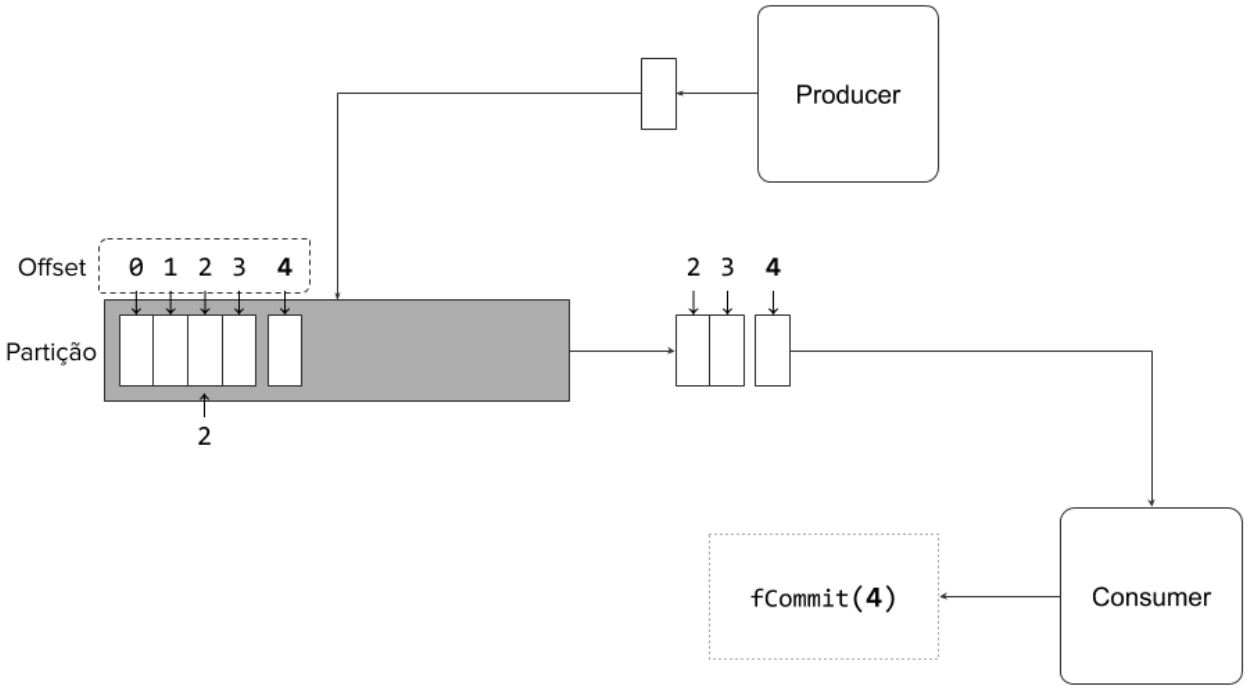

Figura 29: Commit do offset processado

1 - Automático

Podemos dizer que esta é a estratégia menos indicada se existem requerimentos de não reprocessamento, mas ela é a padrão quando não informamos qual desejamos utilizar.

Através da configuração `enable.auto.commit`, que por padrão é `true`, o commit automático torna-se ativo e um processo separado confirma no intervalo definido por `auto.commit.interval.ms`, que o padrão é igual a 5 segundos.

Isso significa que a cada 5 segundos o maior offset consumido será confirmado no Kafka, mas isso pode gerar algumas situações inesperadas.

Exemplo:

- Foram consumidos 50 registros, representados pelo maior offset igual a 230
- Consumidor levará 15 segundos para processá-los
- Após os primeiros 5 segundos, o offset 230 é confirmado
- Após 10 segundos processando, o consumer falha e sua instância é eliminada

- Existe um problema, pois só uma parte dos registros foi processada
- Somente 43 registros foram processados
- Mas todos foram confirmados no offset 230
- Quando tudo voltar ao normal o consumo iniciará a partir do offset confirmado
- 7 registros não serão processados por este grupo de consumo

Então, a melhor opção é sempre configurar `enable.auto.commit` com **`false`**.

2 - Assíncrono

Bem, ao desligar o commit automático será necessário invocá-lo programaticamente. Uma opção é fazê-lo de forma assíncrona, através de um sub-processo que executa em paralelo à linha de execução principal do consumidor.

E apesar de ser eficiente, há chances de falhar. Isso acontece porque o sub-processo não notificará a linha de execução principal, portanto não será possível tomar qualquer ação para correção da falha.

3 - Síncrono

Esta é a melhor abordagem para commit, se você deseja controle sobre o processo de commit. Porque o commit síncrono bloqueia a linha de execução principal do consumidor até que o kafka tenha recebido a confirmação do offset processado.

Ele afeta a taxa de transferência, mas quanto maior for a consistência, menor será a capacidade de processamento da aplicação consumidora. É um *tradeoff*, você precisa decidir entre consistência e taxa de transferência, ou seja, você deverá decidir se aceita uma taxa de transferência menor, para obter uma consistência maior.

Lab 5: consumir dados com base no timestamp

Vamos observar como obter offsets com base em um carimbo data-hora, *timestamp* em inglês, apesar de ser algo pouco comum. E para isso utilizaremos uma implementação feita em Java. Não é necessário que você saiba programar em Java para realizar os experimentos deste lab, basta seguir as instruções.

Se você fez a preparação de sua máquina, tudo o que é necessário para executar este lab já está pronto. Vamos aos passos:

- Faça o download do programa Java
  - Consumer timestamp
- Extraia o conteúdo no diretório `kafka-m1`

Produza dados para testes de consumo

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-lab5-timestamp \
  --num-records 50 \
  --record-size 51200 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      batch.size=307200 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-lab5-timestamp ^
--num-records 50 ^
--record-size 51200 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
batch.size=307200 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

- `51200` é o equivalente a `50KB`
- `307200` é o equivalente a `50KB * 6`

Depois do comando executado com sucesso, além dos registros, também foi criado o tópico `WB-lab5-timestamp`. Ele será utilizado no programa Java para consumo dos dados com base no carimbo data-hora, *timestamp* em inglês.

Consuma os registros para visualizar os carimbos data-hora

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-lab5-timestamp \
  --property print.key=true \
  --property print.timestamp=true \
  --property print.value=false \
  --from-beginning
```

Windows:


```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-lab5-timestamp ^
--property print.key=true ^
--property print.timestamp=true ^
--property print.value=false ^
--from-beginning
```

Basta pressionar `CTRL + C` para o kafka-console-consumer finalizar.

A saída será algo como o exemplo abaixo. Agora, guarde alguns dos valores listados no `CreateTime` para utilização no programa Java.

```
CreateTime:1587668913043    null
CreateTime:1587668913101    null
CreateTime:1587668913102    null
CreateTime:1587668913102    null
CreateTime:1587668913102    null
CreateTime:1587668913103    null
CreateTime:1587668913103    null
CreateTime:1587668913104    null
CreateTime:1587668913104    null
CreateTime:1587668913105    null
CreateTime:1587668913105    null
CreateTime:1587668913135    null
...
```

Experimento a

Informar um carimbo data-hora válido, ou seja, que exista no tópico.

Exemplo sobre como executar o programa, pois é necessário utilizar um dos carimbos data-hora que foram gerados no seu computador.

Linux:

```
./gradlew run \
 -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
 -Dgrupo=WB-kafkatrain \
 -Dtopico=WB-lab5-timestamp \
 -Dtimestamp=1587668913104
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain ^
-Dtopico=WB-lab5-timestamp ^
-Dtimestamp=1587668913104
```


Será possível visualizar que todos os registros produzidos a partir daquela data-hora serão listados no console do seu sistema operacional.

Experimento b

Executar o programa com carimbo data-hora inválido, ou seja, que não existe no tópico.

Linux:

```
./gradlew run \
 -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
 -Dgrupo=WB-kafkatrain \
 -Dtopico=WB-lab5-timestamp \
 -Dtimestamp=0
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain ^
-Dtopico=WB-lab5-timestamp ^
-Dtimestamp=0
```

Veja que foi utilizado `timestamp=0`, que é inválido. Mas isso não causa problema algum e todos os dados presentes no tópico serão listados.

Principais configurações

Assim como nos produtores, também existem dezenas de configurações nos consumidores. Veremos as principais, com insights sobre seu uso ou combinação com outras configurações.

key.deserializer

Nome de classe que receberá um arranjo (array) de bytes e deserializará para um tipo, como uma String, um número ou um objeto.

Para o Kafka as chaves são meros arranjos de bytes que tem seu tipo definido pelo produtor, então o consumidor deverá respeitar isso, ou seja, a chave foi serializada como número e não faz sentido tentar transformá-la em um texto, pois a representação em bytes é para número, não para texto.

- valor padrão: *não há*
- **requerido**: sempre devemos definir seu valor
- implementações mais usadas e disponíveis no Java
  - Texto: `org.apache.kafka.common.serialization.StringDeserializer`
  - Número: `org.apache.kafka.common.serialization.IntegerDeserializer`

  - Número (longo): `org.apache.kafka.common.serialization.LongDeserializer`
  - Fracionários: `org.apache.kafka.common.serialization.FloatDeserializer`
  - Fracionários (longo): `org.apache.kafka.common.serialization.DoubleDeserializer`

value.deserializer

Nome da classe que transformará o valor, que está em bytes, em um tipo para manipulação. Este tipo, normalmente, é uma classe ou qualquer outra estrutura com definição de campos e seus tipos.

No Kafka é muito comum utilizar Avro como um tipo, e novamente, quem define a serialização é o produtor. Deste modo, o consumidor deverá empregar a transformação igualmente equivalente.

- valor padrão: *não há*
- **requerido**: sempre devemos definir seu valor
- implementações mas usadas e disponíveis no Java
  - Texto: `org.apache.kafka.common.serialization.StringSerializer`

> Se o produtor utilizou `io.confluent.kafka.serializers.KafkaAvroSerializer`, o consumidor deverá utilizar `io.confluent.kafka.serializers.KafkaAvroDeserializer`. Isso é o contrato da serialização.

bootstrap.servers

Lista com nome de host, ou ip, e porta, para brokers kafka separados por vírgula: ,

Essa configuração é utilizada para a descoberta do cluster, por este motivo sempre é melhor fornecer ao menos dois hosts. Assim não teremos problemas caso um deles esteja parado, por exemplo.

- valor padrão: *não há*
- **requerido**: sempre devemos definir seu valor

Exemplo:

```
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092,kafka2.ijo:9094
```

fetch.min.bytes

Quantidade mínima de bytes que devem estar disponíveis para que a requisição de consumo seja respondida pelo broker Kafka. Se esta configuração definir valores altos, fará com que o broker aguarde até que os dados sejam acumulados para responder ao consumidor.

- valor padrão: `1`
- configurações relacionadas

    - `fetch.max.wait.ms`

fetch.max.wait.ms

Tempo máximo em milissegundos que o broker aguardará até que existam dados para então responder ao consumidor. Caso a quantidade máxima de bytes não esteja disponível neste tempo, serão respondidos àqueles presentes ou haverá uma resposta vazia.

- valor padrão: `500`
- configurações relacionadas
    - `fetch.min.bytes`

fetch.max.bytes

Tamanho máximo em bytes que o Kafka deverá enviar ao consumidor em um único poll.

- valor padrão: `52428800`
    - é o equivalente a `50MB`

Este não é um limite máximo, ou seja, poderão ser retornados mais bytes além deste limite.

group.id

Identificador do grupo de consumo ao qual o consumidor pertence. O Kafka utiliza este identificador para gerenciar quantos consumidores estão sobrescritos e principalmente os offsets confirmados (*commit*) para cada par tópico+partição.

- valor padrão: *não há*

Utilize um valor com significado. Exemplos bons:

- `clientes_crud_backend`
- `contas_clientes_saldo_backend`

Exemplos ruins:

- `app`
- `serviço`
- `6e47f11e-69a8-4389-acc0-7f4fbd14c454`

auto.offset.reset

A estratégia que deverá ser adotada quando não existir offset para o grupo de consumo ou o offset presente no gerenciamento for inválido no broker.

- valor padrão: `latest`
- valores possíveis
  - `earliest`: consumir tópico desde o início
  - `latest`: consumir tópico a partir do offset mais recente na partição
  - `none`: lançará uma exceção caso nenhum offset exista para o grupo de consumo
  - *qualquer outro valor*: lançará um erro no consumidor

> Um offset poderá ser inválido em função do processo de limpeza e compactação.

session.timeout.ms

Tempo máximo em milissegundos que um consumidor poderá deixar de enviar sinais de vida. Depois disso, automaticamente será iniciado o processo de rebalanceamento.

- valor padrão: `10000`
  - é o equivalente a 10 segundos
- configurações relacionadas no broker
  - `group.min.session.timeout.ms`
  - `group.max.session.timeout.ms`

As configurações relacionadas no broker são limitadores para que nenhum consumidor utilize valores muito baixos ou muito além daquilo que foi estabelecido na instalação do cluster.

heartbeat.interval.ms

Tempo mínimo em milissegundos esperado entre os *heart beats*, que são sinais de vida periódicos que o consumer deverá enviar ao broker. Caso contrário o rebalanceamento de partições é iniciado.

- valor padrão: `3000`
- configurações relacionadas
  - `session.timeout.ms`

Caso um valor maior que session.timeout.ms seja configurado, um erro será lançado no consumidor. Uma mensagem como esta será exibida:

```
Heartbeat must be set lower than the session timeout
```


max.partition.fetch.bytes

Número máximo de bytes por partição que serão entregues ao consumidor em um único poll.

- valor padrão: `1048576`
  - é o equivalente a `1MB`

Este não é um limite máximo, ou seja, poderão ser retornados mais bytes por partição.

enable.auto.commit

Habilita a confirmação (commit) automático dos offsets recebidos.

- valor padrão: `true`
  - sempre configure para `false`
- configurações relacionadas
  - `auto.commit.interval.ms`

Só utilize true em casos de uso muito específicos, como no processamento dos logs de aplicação ou métricas de infraestrutura. Se você utiliza Kafka para casos de negócio, nunca habilite commit automático.

client.id

Identificador do consumidor, utilizado como informativo para logs e métricas, que também são importantes para o gerenciamento de cotas.

- valor padrão: *não há*

Utilize um valor com significado. Exemplos bons:

- `clientes.bff`
- `clientes.svc`

Exemplos ruins:

- `app`
- `serviço`
- `6e47f11e-69a8-4389-acc0-7f4fbd14c454`

allow.auto.create.topics

Permite que o kafka crie o tópico que está sendo consumido, se ele não existir. Essa configuração, quando true, somente funcionará se o broker também permitir.

- valor padrão: `true`
- configurações relacionadas no broker
  - `auto.create.topics.enable`

Caso seja definida como `false` e o tópico em questão não exista, será lançado um erro para o consumidor e a subscrição não será realizada. É comum em ambientes corporativos, não existir a auto-criação de tópicos, devido a governança.

Lab 6: entendendo o consumo de dados com uma App Java

Compreender o consumo de dados sempre é melhor através de aplicações, e neste laboratório serão realizados experimentos com uma aplicação escrita em Java que utiliza os Clientes Kafka oficiais do Apache Kafka.

> Não é necessário que você saiba programar em Java para realizar os experimentos deste lab, basta seguir as instruções.

Se você fez a preparação de sua máquina, tudo o que é necessário para executar este lab já está pronto. Vamos aos passos e depois a cada experimento.

- Faça o download do programa Java
  - Consumidor Java
- Extraia o conteúdo no diretório `kafka-m1`

Experimento a

Entendendo a relação entre `fetch.min.bytes` e `fetch.max.wait`.

> Note que estamos tirando proveito da autocriação de tópicos, que está ativa em nosso cluster Kafka, porque não os estamos criando com o comando `kafka-topics`.

Execute a aplicação para consumir dados do tópico `fetch-min-max-teste` e consumir uma quantidade mínima de `25` Kbytes, ou seja, deverão existir no mínimo esta quantidade de bytes ou o broker aguardará o `fetch.max.wait.ms` para responder com aquilo que existe ou haverá uma resposta vazia.

Linux:


```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dgrupo=WB-lab6 \
  -Dtopico='WB-fetch-min-max-teste' \
  -Dcfg='fetch.min.bytes=25600, fetch.max.wait.ms=30000'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-lab6 ^
-Dtopico='WB-fetch-min-max-teste' ^
-Dcfg='fetch.min.bytes=25600, fetch.max.wait.ms=30000'
```

- `30000` é o equivalente a `30s`
- `25000` é o equivalente a `25KB`

Durante a execução da aplicação você verá uma saída como esta:

```
 > Consumidor java
 > > cfg fetch.min.bytes=25600
 > > cfg fetch.max.wait.ms=30000
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
 > > # registros consumidos: 0
 > > > Consumindo: [fetch-min-max-teste-0]
 > > # registros consumidos: 0
```

Produza uma quantidade de dados menor que o `fetch.min.bytes`, neste caso, apenas `200` bytes.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-fetch-min-max-teste \
  --num-records 1 \
  --record-size 200 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-fetch-min-max-teste ^
--num-records 1 ^
--record-size 200 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```


Em nosso programa Java, na linha 32 do arquivo App.java, temos o seguinte:

```
// . . .
    consumer.poll(Duration.ofSeconds(5));
// . . .
```

O destaque é para o poll(Duration.ofSeconds(5)) que é um controle de bloqueio no cliente Kafka. Neste caso está configurado para 5s, ou seja, ao fazer poll por novos registros somente aguardará cinco segundos por registro, caso contrário nada será retornado. Porém este valor não tem relação com a configuração fetch.max.wait.ms, isso quer dizer que são controles distintos.

Tanto o são, que mesmo após produzir o registro ainda serão vistos alguns retornos vazios no console até que o fetch.max.wait.ms ou fetch.min.bytes sejam atingidos, aquele que acontecer primeiro. Como isso fica claro que fetch.max.wait.ms e fetch.min.bytes são configurações totalmente gerenciadas pelo broker e mesmo que o consumidor envie requisições a cada cinco segundos, o kafka responderá somente quando for o momento correto.

Experimento b

Entendendo a relação entre fetch.min.bytes e fetch.max.wait. ***(continuação)***

Ainda com nossa aplicação Java em execução, vamos produzir registros que no total de bytes serão maiores que fetch.min.bytes.

Linux:

```
kafka-producer-perf-test.sh \
  --topic WB-fetch-min-max-teste \
  --num-records 10 \
  --record-size 10240 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      batch.size=122880 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic WB-fetch-min-max-teste ^
--num-records 10 ^
--record-size 10240 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
batch.size=122880 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

- 10240 é o equivalente a 10KB
- 10 * 10KB = 100KB

- `122880` é o equivalente a `120KB`

Assim que a produção de registros é feita, eles são imediatamente consumidos. Isso aconteceu porque a quantidade de bytes disponíveis era superior àquele definido no `fetch.min.bytes`, portanto não foi necessário aguardar o tempo definido no `fetch.max.wait.ms`.

Experimento c

Entendendo a relação entre as configurações`heartbeat.interval.ms` e `session.timeout.ms`.

Desta vez vamos criar um tópico para que possamos definir o número de partições, pois iniciaremos dois consumidores no mesmo grupo de consumo.

Crie o tópico `WB-heartbeat-lab6`.

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 3 \
  --topic WB-heartbeat-lab6
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 3 ^
--topic WB-heartbeat-lab6
```

Inicie o primeiro consumidor com nosso programa Java que foi utilizado no experimento anterior, só modificando alguns parâmetros.

Consumidor 1 com as configurações padrão.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dgrupo=WB-kafkatrain-l6 \
  -Dtopico='WB-heartbeat-lab6' \
  -Dpoll='5000' \
  -Dcfg='client.id=consumidor-1'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain-l6 ^
-Dtopico='WB-heartbeat-lab6' ^
-Dpoll='5000' ^
-Dcfg='client.id=consumidor-1'
```


Consumidor 2 com modificações na configuração para que seja possível visualizar seu comportamento com relação ao `heartbeat.interval.ms` e `session.timeout.ms`.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dgrupo=WB-kafkatrain-l6 \
  -Dtopico='WB-heartbeat-lab6' \
  -Dpoll='15000' \
  -Dcfg='client.id=consumidor-2, heartbeat.interval.ms=5999, session.timeout.ms=6000'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain-l6 ^
-Dtopico='WB-heartbeat-lab6' ^
-Dpoll='15000' ^
-Dcfg='client.id=consumidor-2, heartbeat.interval.ms=5999, session.timeout.ms=6000'
```

Com essa configuração no Consumidor 2 será possível observar que a todo momento o rebalanceamento será disparado porque os sinais de vida (heartbeats) não estão chegando a tempo no broker.

Agora vamos parar o Consumidor 2 teclando `CTRL + C` e inicie-o novamente, modificando apenas o parâmetro `-Dpoll` para `5000`.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dgrupo=WB-kafkatrain-l6 \
  -Dtopico='WB-heartbeat-lab6' \
  -Dpoll='5000' \
  -Dcfg='client.id=consumidor-2, heartbeat.interval.ms=5999, session.timeout.ms=6000'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain-l6 ^
-Dtopico='WB-heartbeat-lab6' ^
-Dpoll='5000' ^
-Dcfg='client.id=consumidor-2, heartbeat.interval.ms=5999, session.timeout.ms=6000'
```

Observaremos que o rebalanceamento não acontecerá. É assim porque, a cada *poll* que o consumer executa também, são enviados sinais de vida, não ficará só a cargo da configuração destinada a isso.

Pare novamente o Consumidor 2 teclando `CTRL + C` e inicie-o modificando a configuração `heartbeat.internal.ms=2000` e o parâmetro poll para `15000`.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dgrupo=WB-kafkatrain-l6 \
  -Dtopico='WB-heartbeat-lab6' \
  -Dpoll='15000' \
  -Dcfg='client.id=consumidor-2, heartbeat.interval.ms=2000, session.timeout.ms=6000'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dgrupo=WB-kafkatrain-l6 ^
-Dtopico='WB-heartbeat-lab6' ^
-Dpoll='15000' ^
-Dcfg='client.id=consumidor-2, heartbeat.interval.ms=2000, session.timeout.ms=6000'
```

Agora, mesmo com o *poll* a cada `15s`, não haverá rebalanceamento. Porque os sinais de vida serão enviados a cada `2s`. Isso acontece porque o consumer mantém um processo separado só para enviá-los ao Kafka.

Transações

Kafka a partir da versão `0.11.0` introduziu uma API para transações, que não se trata de transações ACID, mas elas garantem que, ou todos os registros serão produzidos, ou nenhum.

Em linhas gerais, as transações são indicadas para casos de uso onde serão produzidos eventos em diversos tópicos e partições diferentes. Assim existirá a garantia de que exatamente todos serão persistidos no Kafka em caso de sucesso, ou exatamente nenhum, em caso de erro.

Apesar das transações fazerem parte da API Producer, também é possível utilizá-la em conjunto com consumidores. Com isso é possível alcançar o padrão de entrega exatamente-uma-vez também no consumidor.

Produtores Idempotentes

Através da propriedade enable.idempotence que está presente no produtor, habilita-se a produção idempotente de registros. Ela garante que os registros destinados a cada partição serão persistidos apenas uma vez, mesmo que várias requisições idênticas sejam enviadas ao Kafka.

Isso é possível graças ao identificador de requisição criado pelo cliente kafka antes de enviar os registros. Este id é gerenciado pelo líder da partição e ao receber requisições com o mesmo, somente manterá a persistência inicial, evitando a duplicidade.

Exemplificando:

- produtor cria id 5 para uma requisição e a envia ao broker;
- produtor aguarda acks all;
- líderes das partições recebem as requisições e persistem os registros;
- existem dois líderes: líder partição #0 e líder partição #3
- ambos os líderes persistem os registros
- alguma instabilidade na rede faz com que o ack da partição #3 não chegue a tempo
- produtor identifica o timeout e inicia o retry
- produtor envia novamente a requisição com id 5
- produtor aguarda acks all
- líderes das partições recebem as requisições e identificam que já foram processadas com sucesso
- líderes respondem com sucesso

Através desse exemplo é possível perceber que o controle da idempotência é em nível de partição, ou seja, se uma requisição possui registros destinados a diversas partições, a garantia de não-duplicidade será controlada isoladamente.

Essa funcionalidade sozinha, já traz muitos ganhos. E em conjunto com as transações, acontecerá um controle global e atômico da produção de eventos em múltiplas partições.

Relação da transação com consumer e a producer

Para tirar o máximo de proveito das transações, pode-se utilizá-las com o consumo de dados. Isso acontece através de um ciclo conhecido como read-process-write, onde consomem-se registros, faz-se o processamento e escreve-se o resultado.

Tudo isso acontecendo em um processo atômico, que efetiva os offsets consumidos e os registros produzidos. E se algo de errado acontecer basta abortar a transação e nada será efetivado, ou seja, offsets não serão confirmados e registros produzidos não serão persistidos.

Existem duas configurações requeridas para trabalhar com contexto transacional.

transaction.timeout.ms

Tempo máximo que o coordenador aguardará pela atualização da transação que deverá partir do produtor, antes que ele seja abordado.

- valor padrão: `60000` (60 segundos)
- configurações relacionadas no broker
  - `transaction.max.timeout.ms`

Se o valor definido for maior que o máximo definido na configuração do broker, que são 15 minutos por padrão, um erro será lançado e não será possível iniciar a produção de eventos.

transactional.id

- valor padrão: *não há*

O identificador da transação deverá ser escolhido com cuidado, porque ao utilizar um valor que está em uso por outro contexto, ocorrerá a seguinte situação:

- quando iniciar uma transação com id `tx800`, o kafka irá eliminar qualquer transação já em andamento com este mesmo identificador
- esta ação do kafka é conhecida como *fence*
- produtores que estão em contexto transacional que passaram por *fence*, receberão ProducerFencedException ao fazer *commit* ou *abort* da transação.

Portanto, escolha com cuidado. Aqui seguem algumas dicas:

- O identificador deverá ser único, mas não dinâmico.
- Evite usar UUID v4
- Ele deverá ser único dentro do cluster kafka
- Ele deverá remeter a aplicação que o está utilizando

Exemplos bons são:

- `<group.id>_<nome-app>`
  - `vendas_usvc-estoque`

Configurações no Consumer

Nada de especial será necessário no consumer, mas seguem algumas notas sobre a configuração `isolation.level`.

Esta configuração tem o valor padrão igual a `read_uncommitted`, sendo possível configurar `read_committed`. Ela tem relação direta com o contexto transacional, portanto seu uso é altamente recomendado, ou seja, seu valor deverá ser `read_committed`.

Lab 7: entendendo transações com uma App Java

O uso de transações só é possível através da Producer API, então, através de aplicação Java, este laboratório mostrará os principais detalhes sobre como escrever um produtor e consumidor.

> Não é necessário que você saiba programar em Java para realizar os experimentos deste lab, basta seguir as instruções.

Se você fez a preparação de sua máquina, tudo o que é necessário para executar este lab já está pronto. Vamos aos passos e depois a cada experimento.

- Faça download do programa Java
  - Producer c/ transação
- Extraia o conteúdo no diretório `kafka-m1`

De forma geral, é como um produtor kafka comum, exceto pelas configurações adicionais. Mas muda antes de iniciar a produção de registros, e também se a implementação consome o kafka, processa e produz registros de volta no kafka. Este é um cenário de uso bastante comum, mas nada impede que seja usado onde exista somente a produção de registros no kafka.

> Porém, se o caso-de-uso não produz múltiplos registros que estão relacionados ao mesmo contexto de execução, somente o produtor idempotente é suficiente.

Experimento a

Como sempre, seu pequeno cluster kafka deverá estar funcionando antes de iniciar este experimento. E não é necessário nada de especial no broker ou no tópico para se possa operar com transações.

Esta aplicação exemplo, consome registros de um tópico de entrada e os produz em outro, sendo este um cenário típico para uso de transações que é conhecido como *read-process-write*.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dconsumir='WB-meu-topico-in' \
  -Dproduzir='WB-meu-topico-out
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dconsumir='WB-meu-topico-in' ^
-Dproduzir='WB-meu-topico-out
```

Então, produza alguns registros no tópico `WB-meu-topico-in`.

Linux:

```
kafka-producer-perf-test.sh \
  --topic 'WB-meu-topico-in' \
  --num-records 20 \
  --record-size 30 \
  --throughput -1 \
  --producer-props \
      acks=1 \
      bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

Windows:

```
kafka-producer-perf-test.bat ^
--topic 'WB-meu-topico-in' ^
--num-records 20 ^
--record-size 30 ^
--throughput -1 ^
--producer-props ^
acks=1 ^
bootstrap.servers=b0.kafka.ml:9092,b1.kafka.ml:9092
```

E verifique o tópico `WB-meu-topico-out` para visualizar os registros que foram produzidos lá.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic 'WB-meu-topico-out' \
  --property print.key=true \
  --property print.timestamp=true \
  --from-beginning
```

Windows:


```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic 'WB-meu-topico-out' ^
--property print.key=true ^
--property print.timestamp=true ^
--from-beginning
```


Trabalhando com Json e Avro

Os formatos de dados mais comuns quando se trabalha com Apache Kafka são Json e Avro. Json é mais fácil de usar por que não requer uma parte móvel adicional.

Quando trabalhamos com Avro é necessário utilizar uma parte móvel chamada Schema Registry para armazenar e versionar o esquema de tipos.

Exemplo Json

Este é o exemplo de uma estrutura JSON para representar um formato de dados complexo.

```
{
  "id":"3524b520-7797-4351-a40d-bfaa77d5d1ba",
  "clienteId":"29961058970",
  "valor": 235.99
}
```

Veja que no JSON os nomes dos campos seguem com os dados e não é mandatório prover um esquema, que por definição, este é um formato de dados com tipagem dinâmica. Porém, existe o JSON-Schema para definir um sistema de tipos para nossas estruturas.

Lab 8: produzir e consumir Json

Vamos entender os detalhes sobre como trabalhar com o formato de dados JSON no Apache Kafka.

> Não é necessário que você saiba programar em Java para realizar os experimentos deste lab, basta seguir as instruções.

Se você fez a preparação de sua máquina, tudo o que é necessário para executar este lab já está pronto. Vamos aos passos e depois a cada experimento.

- Faça download do programa Java
  - Kafka com JSON
- Extraia o conteúdo no diretório `kafka-m1`

Experimento a

Crie o tópico `WB-registros_json`

Linux:


```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 7 \
  --topic WB-registros_json
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 7 ^
--topic WB-registros_json
```

Inicie um console consumer.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-registros_json \
  --property print.key=false \
  --property print.timestamp=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-registros_json ^
--property print.key=false ^
--property print.timestamp=true ^
--from-beginning
```

Execute o programa Java, que produzirá registros no formato de dados JSON.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dtopico='WB-registros_json'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dtopico='WB-registros_json'
```

Analise o código fonte da aplicação Java.

Exemplo de Schema Avro

Neste exemplo Avro existe e definição da estrutura do formato de dados complexo. É mandatório definir o esquema, porque o registro é serializado sem incluir qualquer informação sobre os campos e seu sistema de tipos.

Ao usar Avro, o esquema sempre está separado dos dados e normalmente é armazenado em uma parte móvel chamada Schema Registry.

```
{
  "type":"record",
  "namespace":"com.kafkabr",
  "name":"OrdemCompraFaturada",
  "doc":"Ordem de compra que já foi faturada",
  "fields":[
    {
      "name":"id",
      "doc":"Identificador único da ordem de compra",
      "type":"string"
    },
    {
      "name":"cliente_id",
      "doc":"Identificador único do cliente que criou a ordem de compra",
      "type":"string"
    },
    {
      "name":"valor",
      "doc":"Valor faturado para esta ordem de compra",
      "type":"double"
    }
  ]
}
```

Lab 9: produzir e consumir Avro

Vamos entender os detalhes sobre como trabalhar com o formato de dados Avro no Apache Kafka.

> Não é necessário que você saiba programar em Java para realizar os experimentos deste lab, basta seguir as instruções.

Se você fez a preparação de sua máquina, tudo o que é necessário para executar este lab já está pronto. Vamos aos passos e depois a cada experimento.

- Faça download do programa Java
  - Kafka com Avro
- Extraia o conteúdo no diretório `kafka-m1`

Experimento a

Crie o tópico `WB-registros_avro`

Linux:

```
kafka-topics.sh --create \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --replication-factor 3 \
  --partitions 7 \
  --topic WB-registros_avro
```

Windows:

```
kafka-topics.bat --create ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--replication-factor 3 ^
--partitions 7 ^
--topic WB-registros_avro
```

Inicie um console consumer.

Linux:

```
kafka-console-consumer.sh \
  --bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 \
  --topic WB-registros_avro \
  --property print.key=false \
  --property print.timestamp=true \
  --from-beginning
```

Windows:

```
kafka-console-consumer.bat ^
--bootstrap-server b0.kafka.ml:9092,b1.kafka.ml:9092 ^
--topic WB-registros_avro ^
--property print.key=false ^
--property print.timestamp=true ^
--from-beginning
```

Execute o programa Java, que produzirá registros no formato de dados JSON.

Linux:

```
./gradlew run \
  -Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 \
  -Dtopico='WB-registros_avro'
```

Windows:

```
.\gradlew.bat run ^
-Dkafka=b0.kafka.ml:9092,b1.kafka.ml:9092 ^
-Dtopico='WB-registros_avro'
```


Analise o código fonte da aplicação Java.

Apêndice I

Iniciando um cluster kafka

Notas sobre Kafka no Windows

No windows um cluster kafka não irá executar muito bem, isso em função das limitações existentes no próprio windows. Existe uma issue que busca adaptar o Apache Kafka para rodar em servidores windows, mas ainda não foi aceita:

- https://github.com/apache/kafka/pull/3283

Download

- Crie o diretório `kafka-m1` no seu computador
- Realize o download do arquivo `kafka_2.12-2.4.0.zip`, disponível através do link abaixo:

Download Apache Kafka 2.4.0

- Extraia o conteúdo do arquivo `kafka_2.12-2.4.0.zip` no diretório `kafka-m1`
  - **Linux**: `unzip kafka_2.12-2.4.0.zip`
- Configure a variável de ambiente `KAFKA`.
  - **Linux**: `export KAFKA=/path/to/kafka-m1/kafka_2.12-2.4.0`

Iniciar o zookeeper

- Abra um novo terminal **Linux**:
- Execute o comando:

```
zookeeper-server-start.sh $KAFKA/config/zookeeper.properties
```

- Abra o prompt de comando **Windows**:
- Execute o comando:

```
zookeeper-server-start.bat %KAFKA%\config\zookeeper.properties
```

Quando você observar resultados similares a estes, o zookeeper já está pronto:


```
[2020-03-22 20:50:26,772] INFO binding to port 0.0.0.0/0.0.0.0:2181
    (org.apache.zookeeper.server.NIOServerCnxnFactory)
[2020-03-22 20:50:26,802] INFO zookeeper.snapshotSizeFactor = 0.33
    (org.apache.zookeeper.server.ZKDatabase)
[2020-03-22 20:50:26,805] INFO Snapshotting: 0x0 to /tmp/zookeeper/version-2/snapshot.0
    (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2020-03-22 20:50:26,808] INFO Snapshotting: 0x0 to /tmp/zookeeper/version-2/snapshot.0
    (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2020-03-22 20:50:26,839] INFO Using checkIntervalMs=60000 maxPerMinute=10000
    (org.apache.zookeeper.server.ContainerManager)
```

Iniciar o Kafka Broker #1

- Abra um novo terminal **Linux**
- Digite o comando

```
kafka-server-start.sh $KAFKA/config/server.properties \
  --override broker.id=1 \
  --override log.dirs=/tmp/broker1-logs \
  --override listeners=PLAINTEXT://:9092 \
  --override zookeeper.connect=localhost:2181 \
  --override zookeeper.connection.timeout.ms=10000
```

- Abra o prompt de comando **Windows**:
- Execute o comando:

```
kafka-server-start.bat %KAFKA%\config\server.properties ^
--override broker.id=1 ^
--override log.dirs=/tmp/broker1-logs ^
--override listeners=PLAINTEXT://:9092 ^
--override zookeeper.connect=localhost:2181 ^
--override zookeeper.connection.timeout.ms=10000
```

Iniciar o Kafka Broker #2

- Abra um novo terminal **Linux**:
- Digite o comando:

```
kafka-server-start.sh $KAFKA/config/server.properties \
  --override broker.id=2 \
  --override log.dirs=/tmp/broker2-logs \
  --override listeners=PLAINTEXT://:9093 \
  --override zookeeper.connect=localhost:2181 \
  --override zookeeper.connection.timeout.ms=10000
```

- Abra o prompt de comando **Windows**:
- Execute o comando:


```
kafka-server-start.bat %KAFKA%\config\server.properties ^
--override broker.id=2 ^
--override log.dirs=/tmp/broker2-logs ^
--override listeners=PLAINTEXT://:9093 ^
--override zookeeper.connect=localhost:2181 ^
--override zookeeper.connection.timeout.ms=10000
```

Iniciar o Kafka Broker #3

- Abra um novo terminal **Linux**:
- Digite o comando:

```
kafka-server-start.sh $KAFKA/config/server.properties \
  --override broker.id=3 \
  --override log.dirs=/tmp/broker3-logs \
  --override listeners=PLAINTEXT://:9094 \
  --override zookeeper.connect=localhost:2181 \
  --override zookeeper.connection.timeout.ms=10000
```

- Abra o prompt de comando **Windows**:
- Execute o comando:

```
kafka-server-start.bat %KAFKA%\config\server.properties ^
--override broker.id=3 ^
--override log.dirs=/tmp/broker3-logs ^
--override listeners=PLAINTEXT://:9094 ^
--override zookeeper.connect=localhost:2181 ^
--override zookeeper.connection.timeout.ms=10000
```

Saber se o broker está funcionando

Ao iniciar os brokers, todos eles deverão apresentar a seguinte mensagem, com exceção do id que deverá ser diferente em cada um deles:

```
[2020-03-22 20:54:33,577] INFO [KafkaServer id=3] started (kafka.server.KafkaServer)
```

Problemas comuns no Windows

Para usuários do windows, alguns problemas podem surgir com relação à instalação do Java.

```
Error: missing `server' JVM at `C:\Program Files (x86)\Java\jre1.8.0_151\bin\server\jvm.dll'.
Please install or use the JRE or JDK that contains these missing components.
```

Se você receber o erro acima ao executar qualquer um dos comandos, sua instalação Java trata-se somente da JRE. Mas Kafka requer a JDK, que é a versão completa da Java Virtual Machine.

Solução: revisar a instalação java e utilizar os links disponibilizados na preparação deste módulo.

Docker

Se você tem Docker e Docker Compose instalados na sua máquina, também poderá iniciar o Kafka de forma muito simples.

Cluster c/ Três Brokers

- Baixar o arquivo: Docker Compose Cluster
- Abra um novo terminal Linux e vá até ao diretório onde você baixou o arquivo:
- Digite o comando:

```
docker-compose up
```

Como se trata de um cluster, as portas para os brokers e zookeeper são as seguintes:

- Utilize como bootstrap server `localhost:29092,localhost:39092`
- Zookeeper: `localhost:12181`

Single Broker

- Baixar o arquivo: Docker Compose Single Broker
- Abra um novo terminal Linux e vá até ao diretório onde você baixou o arquivo:
- Digite o comando:

```
docker-compose up
```

