BIG DATA

AULA 4

Prof. Luis Henrique Alves Lourenço

CONVERSA INICIAL

Nesta aula, será apresentado a você um panorama sobre as técnicas e ferramentas para auxiliar no processamento de fluxos de dados. Iniciaremos explorando os principais conceitos que envolvem o processamento de fluxos de dados. Em seguida, vamos nos aprofundar nas principais ferramentas e plataformas de processamento de fluxos de dados distribuídos e ingestão de dados: Kafka, Flume, Storm e Flink.

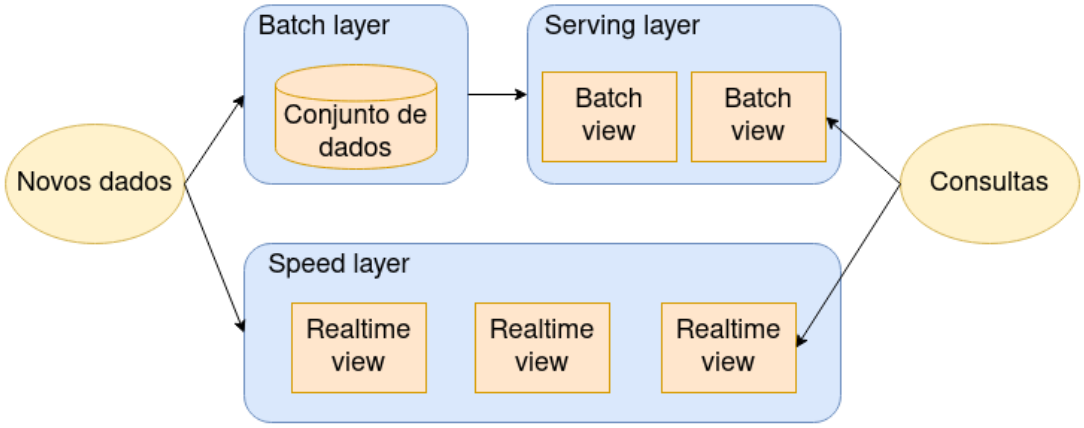
TEMA 1 – PROCESSAMENTO DE FLUXOS DE DADOS DISTRIBUÍDOS

O processamento de fluxos de dados em big data, também chamado de *stream processing*, é uma tecnologia muito importante para processar e analisar dados que são constantemente gerados em tempo real. Diferentemente do processamento tradicional, que exige o armazenamento antes do processamento dos dados, os sistemas de big data estão evoluindo para ser cada vez mais orientados pelo processamento de fluxo de dados, permitindo inserir dados assim que são gerados em ferramentas de análise e obter resultados instantaneamente. Dessa forma, é possível atender à demanda por ferramentas analíticas escaláveis e com baixa latência. Os dados recebidos são tratados na chegada, e os resultados são atualizados de forma incremental enquanto os dados são processados continuamente pelo sistema. Uma vez que não é possível acessar todo o fluxo de dados de uma única vez, o processamento se dá em janelas baseadas em tempo ou *buffer* sobre os dados recém-chegados em pequenas computações independentes.

1.1 A arquitetura lambda

Os sistemas de processamento de big data modernos tentam combinar o processamento em lote (*batch*) com o processamento de fluxos de dados em um ou mais *pipelines* paralelos. Marz e Warren (2015) propõem a arquitetura lambda, que inclui o processamento em lote e o processamento de fluxo de dados em um mesmo sistema de processamento de big data. Essa arquitetura é projetada para aplicações que possuem atrasos na coleção de dados e processamento devido ao uso de interfaces interativas ou *dashboards* e passos de validação de dados.

Figura 1 – Arquitetura lambda



Fonte: Marz; Warren, 2015.

A ideia principal da arquitetura lambda é construir um sistema de big data em camadas em que cada camada é responsável por um subconjunto de propriedades. Segundo Marz e Warren (2015), as camadas da arquitetura lambda são: *speed layer*, *serving layer* e *batch layer*. Idealmente pode-se executar funções a qualquer momento para obter resultados. Infelizmente, mesmo sendo possível, esse tipo de execução pode necessitar de uma grande quantidade de recursos, uma vez que pode ser necessário acessar *petabytes* de dados. Dessa forma, a *Batch layer* armazena todo o conjunto de dados e pré-processa uma quantidade de consultas em forma de *views* (visualizações) para serem acessadas instantaneamente.

O próximo passo é incorporar as *views* em alguma interface onde elas possam ser consultadas. A *serving layer* é uma base de dados distribuída especializada em carregar as *views* geradas pelas demais camadas. Até esse ponto satisfazem as propriedades desejadas de um sistema de big data definidos por Marz e Warren (2015). São elas:

- **Robustez e tolerância a falhas**: refere-se à capacidade dos sistemas se recuperarem de falhas que possam fazer com que os servidores do *cluster* em questão possam ficar indisponíveis;
- **Escalabilidade**: refere-se à capacidade de atender ao processamento de grandes volumes;
- **Generalização**: o uso dessa arquitetura é projetado para computar e atualizar *views* de conjuntos de dados arbitrários;

- **Extensibilidade**: refere-se à facilidade em adicionar novas *views* e novos tipos de dados. Novas *views* podem ser adicionadas por meio da criação de novas funções para processar o conjunto de dados;
- **Consultas Ad Hoc**: os dados contidos no cluster podem ser acessados diretamente pela *batch layer* a qualquer momento;
- **Manutenção mínima**: o principal componente a ser mantido é o *Hadoop*, que necessita de algum conhecimento de administração, mas é suficientemente simples de operar. Enquanto que a *serving layer* é ainda mais simples e, portanto, possui menos componentes que possam falhar;
- ***Debuggability***: as entradas e resultados do sistema devem gerar informação necessária para identificar problemas no sistema ou nas funções escritas para o sistema.

Dessa forma, as camadas descritas são capazes de atender a tais propriedades em quase sua totalidade, porém uma única propriedade não atendida por tais sistemas é a atualização das *views* com baixa latência. Para isso, a arquitetura lambda prevê a implementação da *speed layer*. A *speed layer* é a camada que implementa o processamento de fluxo de dados com o objetivo de superar a latência do processamento de grandes volumes de dados e oferecer percepções úteis antes de armazená-las. Nessa camada, os dados são processados de forma incremental, escalável e tolerante a falhas tão logo são recebidos. O processamento de fluxos de dados se mostra competitivo em um cenário em que são gerados em tempo real e o valor da informação contida nesses dados decresce rapidamente com o tempo. Por exemplo, em sistemas de monitoramento, de negócios financeiros, entre muitos outros.

Enquanto a *serving layer* é atualizada à medida que a *batch layer* finaliza o pré-processamento de suas *views*, os dados que são recebidos não estão sendo incluídos em seu processamento. Para atender a essa demanda, é necessário haver um sistema que processa os dados em tempo real. Dessa forma, a *speed layer* tem como objetivo possibilitar que os novos dados sejam representados o quanto antes pelas funções de consulta assim que necessário pelos requisitos da aplicação. A *speed layer* produz *views* de forma semelhante a *batch layer*. A principal diferença é que a *speed layer* apenas processa os dados recentes enquanto a *batch layer* processa todos os dados de uma vez. Além disso, a *speed layer* faz a computação incremental dos dados em vez da recomputação dos dados feita pela *batch layer*.

Uma vez que os dados sejam processados pela *batch layer* e cheguem até a *serving layer*, o resultado correspondente na *speed layer* pode ser descartado. Descartar as *views* que já não são mais necessárias é importante pois a *speed layer* é muito mais complexa que a *batch layer* e a *serving layer*. Essa propriedade é conhecida por *isolamento de complexidade* e significa que a complexidade deve ser contida nos componentes temporários da arquitetura. Se algo falhar, é possível descartar as *views* da *speed layer* e em pouco tempo tudo voltará ao funcionamento normal.

1.2 Ferramentas de ingestão de dados

Grande parte da importância da arquitetura lambda se encontra na necessidade de organizar o fluxo de entrada de dados de acordo com a exigência de latência do sistema. Dessa forma, é possível utilizar um conjunto de ferramentas para gerenciar a entrada dos dados, distribuí-los pelos componentes do sistema na ordem correta, coletar resultados, escalonar os componentes para evitar que fiquem sobrecarregados e gerenciar falhas. Nos próximos temas desta aula, vamos explorar algumas dessas ferramentas.

TEMA 2 – KAFKA

Kafka é uma plataforma de processamento de fluxos de dados desenvolvido e mantido pela Fundação Apache e implementada utilizando as linguagens Java e Scala. O principal objetivo do Kafka é oferecer uma plataforma unificada de alta capacidade e baixa latência para o processamento de dados em tempo real. Foi projetada como uma plataforma para ser utilizada por diversos sistemas, não apenas sistemas Hadoop. Seu funcionamento é baseado em uma fila de mensagens de propósito geral maciçamente escalável e distribuída. Como uma plataforma de processamento de fluxos, o Kafka possui 3 características centrais:

- A capacidade de publicar e se inscrever em fluxos de eventos de forma semelhante a um sistema de filas de mensagens;
- O armazenamento de fluxos de eventos de uma forma durável e tolerante a falhas;
- O processamento de fluxos de eventos assim que eles ocorram.

Como uma plataforma distribuída, o Kafka é executado em um *cluster* de um ou mais servidores que podem abranger múltiplos *datacenters*. Um cluster Kafka armazena fluxos de eventos em categorias chamadas *tópicos*. Cada servidor é capaz de manter a última mensagem recebida por consumidor. Isso permite que um fluxo de eventos que tenha sido interrompido possa continuar a sua transmissão sem que haja perda de eventos. Os eventos recebidos são armazenados por determinado tempo e publicados em tópicos. Cada evento consiste em uma chave, um valor e um *timestamp*. Dessa forma, é possível garantir que os eventos que são recebidos sejam entregues na mesma ordem. O cliente pode se inscrever em um ou mais tópicos para receber os eventos tão logo sejam publicados. Cada tópico pode possuir muitos clientes inscritos como consumidores. Dadas essas características, o Kafka pode ser utilizado tanto como um sistema de mensagens, um sistema de armazenamento, quanto como um sistema de processamento de fluxos de dados.

2.1 Kafka como um sistema de mensagens

Os sistemas de mensagens tradicionalmente podem seguir tanto o modelo de fila de mensagens quanto o modelo de publicação-inscrição. No modelo de fila de mensagens, um conjunto de consumidores pode ler os dados de um servidor, porém a partir do momento em que um deles faz a leitura e recebe os dados, os demais não poderão ler os mesmos dados, pois os dados deixam de existir no servidor. Por outro lado, no modelo de publicação-inscrição, os dados são enviados para todos os consumidores inscritos.

Kafka é implementado de forma que satisfaz ambos os modelos. Assim como uma fila de mensagens, é capaz de dividir o processamento das mensagens em diversos processos que consumirão os dados. E assim como o modelo de inscrição-publicação, também é capaz de enviar uma mesma mensagem para um grupo de consumidores. Dessa forma, os tópicos em Kafka são capazes de reproduzir as propriedades de ambos os modelos, sendo capaz de escalar o processamento e também permitir dividir o envio das mensagens por grupos de inscritos.

2.2 Kafka como um sistema de armazenamento

Toda fila de mensagens que é capaz de publicar mensagens de forma desvinculada com o processo que irá consumi-las age efetivamente como um sistema de armazenamento. Os dados enviados para o Kafka são armazenados em disco e replicados para garantir o princípio da tolerância a falhas. O Kafka permite que os produtores aguardem pela confirmação de que os dados foram armazenados. Isso permite que o envio dos dados seja garantido. Dessa forma, o Kafka pode ser considerado um sistema de arquivos distribuído para propósitos especiais dedicados à alta performance, baixa latência, replicação e propagação.

2.3 Kafka como um sistema de processamento de fluxos de dados

Além de ser capaz de ler, escrever e armazenar fluxos de dados, seu principal propósito é permitir o processamento em tempo real. Um processador de fluxos em Kafka deve receber fluxos contínuos de dados em tópicos, realizar operações de processamento sobre os dados recebidos e produzir um tópico com o fluxo de dados resultantes. O Kafka é capaz de realizar tais operações utilizando suas APIs básicas, mas para operações mais complexas o Kafka oferece a *Streams API*, que permite a criação de aplicações que são capazes de efetuar agregações e *joins* de fluxos de dados.

2.4 Arquitetura

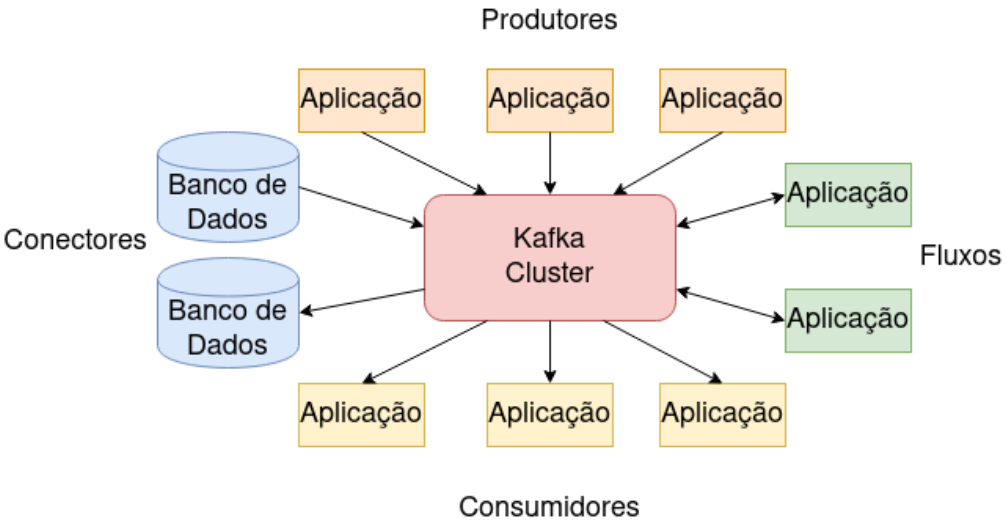
A combinação de um sistema de mensagens, com armazenamento e processamento de fluxos, é essencial para o funcionamento do Kafka como uma plataforma de fluxos. Assim, o Kafka é capaz de processar tanto os dados que ainda chegarão quanto os que estão armazenados. Para isso, o Kafka utiliza uma combinação de 4 APIs:

- A API produtor (P*roducer API*) permite que uma aplicação publique um fluxo de eventos em um ou mais tópicos;
- A API consumidor (C*onsumer API*) permite que uma aplicação se inscreva a um ou mais tópicos e processe o fluxo de eventos produzidos por eles;
- A API de fluxos (*Streams API*) permite a uma aplicação agir como um processador de fluxos, consumindo um fluxo de um ou mais tópicos e

produzindo um fluxo de um ou mais tópicos. Efetivamente transformando tópicos de entrada em tópicos de saída;

- A API conector (*Connector API*) permite criar e utilizar produtores e consumidores reutilizáveis que conectam tópicos Kafka a aplicações existentes ou sistemas de dados. Por exemplo, um conector pode capturar cada mudança realizada em uma tabela de um banco de dados relacional.

Figura 2 – Arquitetura Kafka



Fonte: Kafka Apache, S.d.

A comunicação em Kafka entre os clientes e os servidores é realizada utilizando-se o protocolo TCP. Dessa forma, a comunicação ocorre de forma simples, com alta performance e sem dependência com nenhuma linguagem de programação. O Kafka oferece um cliente Java, mas há clientes em diversas outras linguagens.

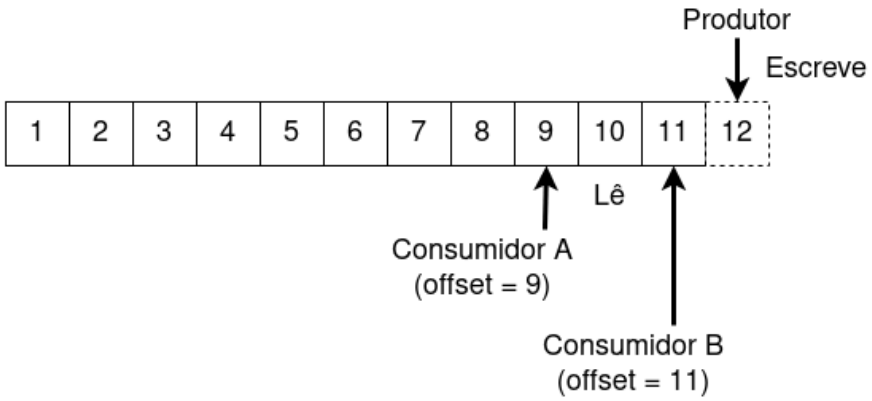
2.5 Tópicos e *logs*

A principal abstração para oferecer um fluxo de eventos em Kafka são os tópicos. Tópico é uma categoria ou *feed* em que os eventos são publicados. Tópicos em Kafka podem possuir muitos inscritos. Para cada tópico, o *cluster* Kafka mantém um *log* particionado. Cada partição é uma sequência ordenada e imutável de eventos que é continuamente acrescentada a um *log* estruturado. As partições em um *log* permitem ao *log* escalar além de um único servidor. Cada partição individualmente deve caber nos servidores onde estão hospedados. As

partições de um *log* são distribuídas pelos servidores de um *cluster* Kafka em que cada servidor manipula os dados e as requisições por uma porção da partição. Cada partição é replicada por meio de um número configurável de servidores de forma a garantir a tolerância a falhas. Porém um tópico deve possuir tantas partições que o permitam manipular uma quantidade arbitrária de dados. Além disso, permitindo atuar como uma forma de paralelismo.

Aos eventos na partição são atribuídas identificações sequenciais conhecidas por *offset*, que identifica individualmente cada evento na partição. Os eventos armazenados em um *cluster* Kafka persistem de forma durável por período configurável – tenham eles sido consumidos ou não. O único metadado retido por consumidor é o *offset* marcando a posição do consumidor no *log*. O *offset* é controlado pelo consumidor, normalmente avançando linearmente à medida que os eventos do *log* são lidos. No entanto, como o *offset* é controlado pelo consumidor, o *log* pode ser consumido em qualquer ordem, por isso um consumidor em Kafka é muito barato.

Figura 3 – Produtor e consumidores em Kafka



Fonte: Kafka Apache, S.d.

2.6 Produtores

Produtores publicam dados dos tópicos de sua escolha. O produtor é responsável por escolher qual evento deve ser atribuído a qual partição em um tópico. Dessa forma, o balanceamento de carga do cluster pode ser dar por meio de um escalonador *round-robin*, ou pode ser realizado por um particionador semântico, ou seja, baseado em alguma característica do evento.

2.7 Consumidores

Consumidores se agrupam em grupos de consumidores, e cada evento publicado a um tópico é entregue a uma instância consumidora dentro da inscrição de um grupo consumidor. Instâncias consumidoras podem ser processos separados em máquinas separadas. Se todas as instâncias consumidoras estiverem em um mesmo grupo consumidor, então os eventos serão balanceados entre todas as instâncias consumidoras. Por outro lado, se cada instância consumidora estiver em um grupo consumidor diferente, então cada evento será enviado para cada instância consumidora. Porém, normalmente, tópicos possuem um número pequeno de grupos consumidores. Cada grupo é composto de várias instâncias consumidoras de forma a beneficiar a escalabilidade e a tolerância a falhas. Isso é, portanto, semântica publicação-inscrição onde cada inscrito é um *cluster* de consumidores em vez de um único processo.

TEMA 3 – FLUME

Flume é um sistema distribuído, confiável e de alta disponibilidade para coletar, agregar e mover eficientemente grandes quantidades de dados de *log* de muitas fontes diferentes para um armazenamento centralizado. Flume é um projeto da Fundação Apache criado originalmente para o Hadoop como um sistema de agregação de *log* com suporte nativo ao HDFS e ao HBase. Atualmente, o Flume não é mais restrito a agregação de *log*. As fontes de dados são customizáveis. Dessa forma, o Flume é adequado para o transporte de uma quantidade massiva de dados de praticamente qualquer fonte possível. Além disso, o Flume permite construir um fluxo em diversas etapas em que os eventos trafegam por múltiplos agentes antes de atingir sua destinação final. Entre as vantagens de seu uso, destaca-se a habilidade de atuar como um tipo de *buffer* quando a taxa de dados recebidos supera a capacidade de escrita do destino. Além disso, o Flume permite realizar o roteamento contextual dos dados.

Eventos são organizados em um canal em cada agente. Então os eventos são entregues ao próximo agente ou a um repositório terminal, como o HDFS. Porém, os eventos são removidos de um canal apenas após serem armazenados pelo canal do próximo agente ou repositório terminal. Dessa forma, a semântica de entrega de mensagens em cada etapa é capaz de garantir

a confiabilidade do fluxo de ponta a ponta. O Flume utiliza uma abordagem transacional para garantir a confiabilidade de entrega de eventos. As fontes e destinos em uma transação encapsulam respectivamente o armazenamento e a obtenção dos eventos localizados ou fornecidos por uma transação em um canal. Assim, é possível garantir que um conjunto de eventos seja passado de forma confiável de um ponto a outro em um fluxo. No caso de um fluxo em diversas etapas, tanto o destino de uma etapa anterior quanto a fonte da próxima etapa mantêm as suas transações executando para garantir que os dados sejam armazenados de forma segura no canal da próxima etapa.

Os eventos estão organizados em canais capazes de se recuperarem de falhas. Flume suporta canais de arquivos duráveis baseados no sistema de arquivos local. Além disso, o Flume também suporta canais que armazenam os dados em filas em memória. Esse tipo de canal é mais rápido, porém qualquer evento que estiver em um canal baseado em memória quando um processo agente é terminado não pode ser recuperado.

3.1 Arquitetura

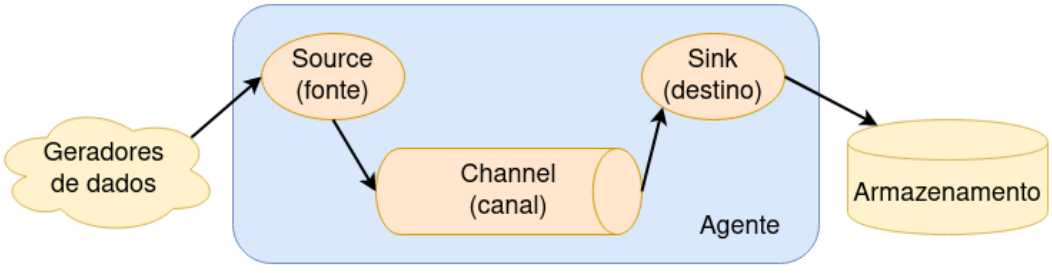
A arquitetura do Flume pode ser resumida da seguinte forma: os dados gerados por diversas aplicações, por meio de clientes Flume, são coletados por agentes Flume, que são compostos de fontes (*sources*), canais (*channels*) e destinos (*sinks*). Podemos identificar esses componentes na Figura 4. Os principais componentes de um sistema Flume são:

- **Eventos:** um evento é uma unidade de dado que pode ser transportada. Normalmente um evento é um registro simples de um conjunto de dados maior. Sua estrutura é composta de um conjunto de cabeçalhos opcionais seguida do conjunto de *bytes* que compõe os dados a serem transportados;
- ***Source* (ou fonte):** é a fonte que recebe e gerencia os dados de aplicações externas e os transfere para um ou mais canais;
- ***Channel* (ou canal):** funciona como um elemento que armazena os eventos das *sources* até que eles sejam consumidos pelos *sinks*. *Channels* também são capazes de ditar a durabilidade dos eventos entregues entre as *sources* e os *sinks*. Os *channels* podem manter os eventos em memória, onde a transferência ocorre de maneira mais rápida,

porém sem garantia contra perda de dados; ou podem manter os dados armazenados de forma durável em arquivos, onde a entrega ao *sink* é garantida, apesar de não conseguir realizar a transferência dos eventos com a mesma velocidade quanto os *channels* que utilizam memória;

- ***Sink* (ou destino):** é a contraparte da *source*. É o componente que gerencia como os dados serão entregues. Consome os eventos de um *channel* e os entrega a seu destino final. Um dos *sinks* incorporados nativamente ao Flume é o HDFS *sink* que permite escrita de eventos diretamente ao sistema de arquivos distribuído do *Hadoop*;
- ***Runners*:** é um componente interno às *sources* e aos *sinks* responsável por sua execução. Normalmente o usuário não precisa tomar conhecimento de sua execução, mas pode ser útil para os desenvolvedores de *sources* e *sinks* entenderem os detalhes de seu funcionamento;
- **Agente:** de forma genérica, é qualquer processo independente (Java Virtual Machine – JVM) executando em um sistema Flume. Um único agente pode conter qualquer quantidade de *sources*, *sinks* e *channels*. É o responsável por receber os dados das aplicações do cliente ou outros agentes e encaminhá-la para o próximo componente. O Flume pode executar mais de um agente;
- **Provedor de configuração:** originalmente a configuração do Flume era gerenciada pelo Zookeeper, porém, em sua versão mais recente, o usuário pode optar entre diversos *plugins* de sistemas de configurações diferentes;
- **Cliente:** não é necessariamente um componente do Flume, porém funciona como um conector que envia os dados a um agente.

Figura 4 – Arquitetura Flume



Fonte: Apache Flume, 2019.

Além disso, vale a pena destacar alguns componentes adicionais que desempenham um papel importante. Os interceptadores são componentes utilizados para adicionar, modificar ou inspecionar dados enquanto são transferidos entre as *sources* e os *channels*. Os *channel selectors* (ou seletores de canal) são componentes utilizados a partir da *source* para determinar qual o canal mais adequado para os eventos serem transferidos em caso de um agente com múltiplos *channels*. Há dois tipos de *channel selectors*: os *default channel selectors*, que replicam um evento para todos os *channels*; e os *multiplexing channel selectors*, que decidem para qual *channel* deve ser enviado o evento baseado no endereço definido no cabeçalho do evento. E, por fim, os *sink processors* são componentes utilizados para selecionar um *sink* de um grupo de *sinks*. Além disso, os *sink processors* podem ser utilizados para definir caminhos alternativos para falhas em *sinks*, ou mesmo para o balanceamento de falha entre os *sinks*.

3.2 Configuração e exemplo

A configuração de agentes Flume é armazenada em um arquivo local. É um arquivo texto que segue o formato de arquivos de propriedades Java. Configurações para um ou mais agentes podem ser especificadas em um mesmo arquivo de configuração. Além disso, são configuradas as propriedades dos componentes de cada agente, como as *sources, channels* e *sinks*; e como cada um desses componentes estão ligados para formar um fluxo de dados. Um arquivo de configuração Flume se parece com o seguinte exemplo:

Figura 5 – Exemplo de arquivo de configuração Flume

```
# exemplo.conf: Configuração de um nó Flume

# Nomeia os componentes deste agente
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Descreve e configura a source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Descreve o sink
a1.sinks.k1.type = logger

# Configura Channel que bufferiza events em memoria
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

# Conecta a source e o sink ao channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

O exemplo acima descreve um agente que recebe eventos de uma conexão *netcat* a partir da porta 44444 da máquina local em um *source*, armazena os dados em memória por meio de um *channel* e escreve os dados no *logger* do terminal por meio do *sink*. Utilizando-se dessa configuração é possível executar o Flume utilizando o seguinte comando:

Figura 6 – Comando para execução do Flume

```
$ bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -
Dflume.root.logger=INFO,console
```

Para completar o exemplo, em um terminal separado, pode-se executar o *telnet* utilizando a porta 44444 para enviar eventos para o Flume:

Figura 7 – Porta 44444

```
$ telnet localhost 44444
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello world! <ENTER>
OK
```

Dessa forma, o terminal que está executando o Flume exibirá as seguintes informações de acordo com a configuração do parâmetro "-Dflume.root.logger=INFO,console", que define que as informações de *log* do sistema Flume sejam retornados no console do terminal:

Figura 8 – Informações de log do sistema Flume

```
12/06/19 15:32:19 INFO source.NetcatSource: Source starting
12/06/19 15:32:19 INFO source.NetcatSource: Created
serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
12/06/19 15:32:34 INFO sink.LoggerSink: Event: { headers:{} body: 48
65 6C 6C 6F 20 77 6F 72 6C 64 21 0D          Hello world!. }
```

Aqui podemos ver os dados que foram escritos no *telnet* passaram pelo agente Flume que foi definido pela nossa configuração e foram escritos no *logger* do terminal.

TEMA 4 – STORM

Storm é um sistema de computação distribuída em tempo real desenvolvido para permitir o processamento de fluxos de dados ilimitados e pode ser utilizado com qualquer linguagem de programação. É muito eficiente, sendo capaz de processar milhões de tuplas por segundo em um único nó. Sua topologia é inerentemente paralela, uma vez que é feita para executar em um *cluster* de máquinas, dessa forma garantindo a escalabilidade horizontal do sistema. Além disso, diferentes partes da topologia podem escalar individualmente por meio de seu paralelismo. Utilizando a linha de comandos do Storm é possível executar o comando *rebalance* para ajustar o paralelismo das topologias que estiverem executando sem precisar parar a execução. Assim, o Storm é capaz de processar mensagens com alto rendimento e baixa latência. É tolerante a falhas, ou seja, quando um processo trabalhador morre, ele é

reiniciado automaticamente. Se um nó inteiro ficar indisponível, os processos trabalhadores são reiniciados automaticamente em um outro nó. Isso permite garantir que todos os dados sejam processados independentemente da ocorrência de falhas no sistema. *Clusters* Storm necessitam de um conjunto mínimo de configuração para iniciar e operar. O Storm foi desenvolvido parar ser muito robusto e pode continuar operando por longíssimos períodos de tempo. O Storm foi projetado com uma arquitetura mestre-trabalhador que utiliza o Zookeeper para coordenar os nós. O nó mestre é chamado de *Nimbus* e os trabalhadores de *supervisores*. Entre os casos de usos do Storm, podemos destacar o seu uso em analítica de dados em tempo real, aprendizado de máquina, computação contínua e muitas outras aplicações. Por meio da abstração *spout*, *o* Storm é capaz de integrar sistemas de mensagens, como Kestrel, RabbitMQ, Kafka, JMS e Amazon Kinesis. E também é capaz de se integrar facilmente com diversas tecnologias de bancos de dados. Além disso, é possível utilizar o framework Thrift para permitir a geração de código utilizando qualquer linguagem de programação para construir serviços escaláveis. Ou seja, é possível escrever definições em Thrift para submeter topologias ou qualquer outro componente do Storm.

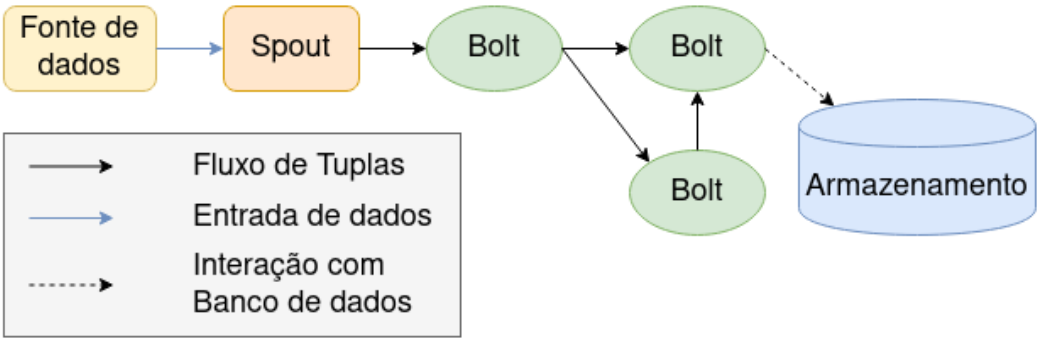
4.1 Topologia e demais conceitos

O Storm possui uma API simples e fácil de utilizar. A programação em Storm se baseia na manipulação e transformação de fluxos de tuplas, que são estruturas que podem conter objetos de qualquer tipo. Além disso, é possível definir um serializador para tipos não existentes no Storm. Existem três tipos principais de abstrações em Storm: *spouts*, *bolts* e topologias.

- **Spout:** um *spout* é uma fonte de fluxos que normalmente recebe tuplas de fontes externas. *Spouts* podem ser configurados para repetirem tuplas em casos de falhas no processamento, ou descartar tuplas assim que são emitidas. Um *spout* pode ler um *broker* de fila de mensagens, como Kestrel, RabbitMQ ou Kafka; gerar seu próprio fluxo; ou ler outras APIs de fluxos. Existem implementações da maioria dos sistemas de filas de mensagens para o Storm. Um *spout* pode emitir mais de um fluxo de tuplas;

- **Bolt:** um *bolt* processa qualquer número de entradas de fluxos, processa seus dados e produz novos fluxos. Um *bolt* pode produzir mais de um fluxo de tuplas. A lógica computacional de um sistema Storm é toda realizada em *bolts* na forma de funções, filtros, uniões de fluxos, agregações de fluxos, comunicando com bancos de dados, entre outras formas;
- **Topologia:** uma topologia é uma rede de *spouts* e *bolts*, em que cada aresta na rede representa um *bolt* ligado a um fluxo de saída de um *spout* ou outro *bolt*. Uma topologia é um fluxo computacional de complexidade arbitrária. Pode ser comparado de forma análoga a uma tarefa MapReduce do Hadoop.

Figura 9 – Exemplo de topologia Storm



Topologias executam em um ou mais processos trabalhadores. Cada processo trabalhador é, em última instância, uma JVM (*Java Virtual Machine*) e executa um subconjunto de todas as tarefas da topologia. O Storm distribui as tarefas por todos os processos trabalhadores. Cada *bolt* ou *spout* executa quantas tarefas sejam necessárias no cluster. Cada tarefa corresponde a uma *thread* e o agrupamento de fluxos define como enviar tuplas de um conjunto de tarefas para outro. É possível configurar manualmente o paralelismo dos *bolts* e *spouts*. O agrupamento de fluxos é parte da definição da topologia de forma a especificar quais fluxos de tuplas devem ser recebidos para cada *bolt*. Cada agrupamento de fluxo define como o fluxo deve ser particionado entre as tarefas de cada *bolt*. Existem 8 tipos de agrupamento de fluxo incorporado nativamente ao Storm:

- ***Shuffle grouping*:** *agrupamento embaralhado*, ou seja, tuplas são distribuídas aleatoriamente às tarefas dos *bolts* de forma que as tuplas sejam distribuídas de maneira igualitária;

- ***Fields grouping*****:** *agrupamento por campo*, ou seja, as tuplas são distribuídas de acordo com determinado campo definido pelas tuplas;
- ***Partial Key grouping*****:** *agrupamento por chave parcial*, significa que as tuplas são agrupadas pela própria configuração do agrupamento, por exemplo levando em consideração o balanceamento de carga do sistema;
- ***All grouping*****:** *agrupamento total*. Nesse caso, o fluxo é replicado para todas as tarefas dos *bolts*;
- ***Global grouping*****:** *agrupamento global*. O fluxo inteiro é direcionado a uma única tarefa do *bolt*. A tarefa de menor identificador é especificamente selecionada;
- ***None grouping*****:** indica que nenhum agrupamento foi selecionado. Atualmente esta opção é idêntica ao *shuffle grouping*;
- ***Direct grouping*****:** *agrupamento direto*. É um tipo especial de agrupamento em que o produtor do fluxo decide qual tarefa do consumidor deve receber cada tupla;
- ***Local or shuffle grouping*****:** se o *bolt* que recebe o fluxo possui um ou mais tarefas no mesmo processo trabalhador, as tuplas são distribuídas de forma embaralhadas apenas as tarefas contidas no mesmo processo trabalhador; caso contrário, esse agrupamento se comporta como um *shuffle grouping* normal.

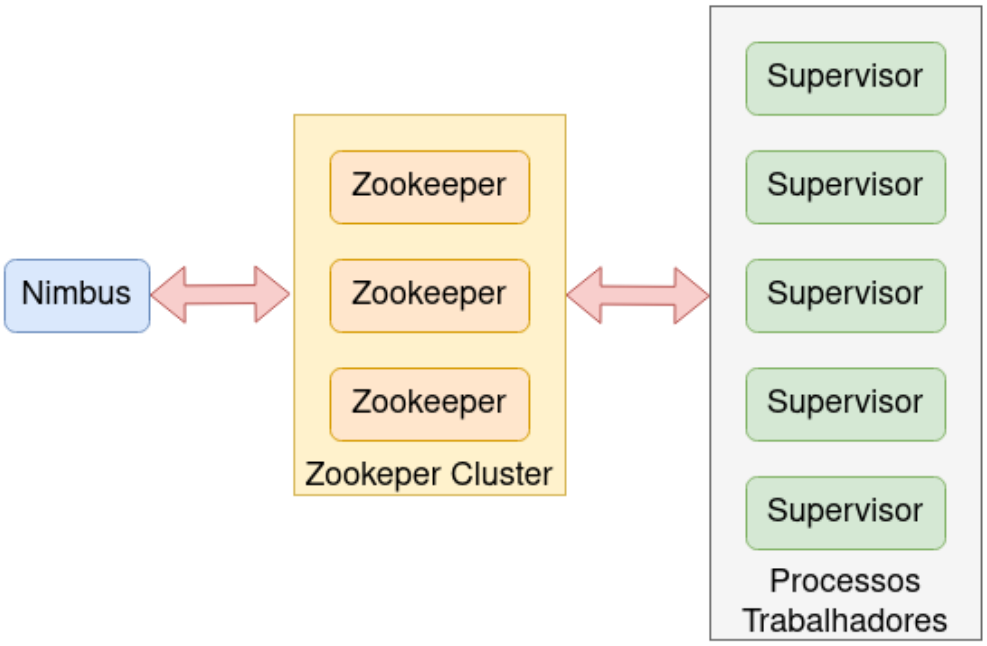
4.2 Arquitetura do *cluster* Storm

Um *cluster* Storm segue o modelo mestre-trabalhador, em que os processos mestre e trabalhadores são coordenados pelo Zookeeper. O nó mestre é conhecido por *Nimbus*, que é responsável por distribuir o código da aplicação em vários nós trabalhadores, atribuindo tarefas a diferentes máquinas, monitorando falhas nas tarefas e reiniciando-as quando necessário. Todos os dados do Nimbus são armazenados pelo Zookeeper. Há um único Nimbus ativo em cada *cluster* Storm. No caso de falha do Nimbus ativo, um novo nó passivo se torna ativo assumindo rapidamente o lugar do Nimbus que falhou. Os trabalhadores continuam trabalhando mesmo que não exista nenhum Nimbus ativo.

Os nós supervisores são os nós trabalhadores em um *cluster* Storm. Cada nó supervisor executa um *daemon* supervisor que é responsável por criar, iniciar

e parar processos trabalhadores que executam as tarefas atribuídas ao nó. Assim como os Nimbus, os *daemons* supervisores são reiniciados rapidamente em caso de falha uma vez que armazenam todos os seus estados no Zookeeper. Um único *daemon* supervisor gerencia um ou mais processos trabalhadores na mesma máquina.

Figura 10 – Arquitetura Storm



Fonte: Apache Software Foundation.

4.3 Interfaces alternativas

Storm possui algumas interfaces alternativas para construir topologias. São elas: Trident, Streams API, Storm SQL e Flux.

4.3.1 Trident

Trident é uma abstração de alto nível que permite realizar computação de tempo real utilizando o Storm. Essa interface permite combinar o processamento de fluxos em alto desempenho e consultas de baixa latência. Trident permite a execução de operações como *join*, agregações, agrupamentos, funções e filtros. Além disso, são incluídas primitivas para o processamento incremental sobre qualquer base de dados ou armazenamento persistente.

4.3.2 Streams API

Originalmente o Storm oferece APIs *spouts* e *bolts* para criar fluxos computacionais. Esses APIs são simples de utilizar, porém não implementam uma forma simples de reutilizar fluxos construídos para operações como filtragem, transformações, *windowing*, *join*, agregação, e muitas outras. Assim, a Stream API é construída com base nas APIs *spouts* e *bolts* de forma a oferecer uma API para construir fluxos computacionais e suporte a operações funcionais como MapReduce.

4.3.3 Storm SQL

A interface StormSQL permite executar consultas SQL sobre fluxos de dados em Storm. Além de permitir ciclos de desenvolvimento em análise de fluxos, uma interface SQL permite a oportunidade de unificar ferramentas de processamento em lote, como o Hive, com a análise de fluxo em tempo real. Em um altíssimo nível, o Storm SQL interpreta consultas SQL convertendo-as em topologias Storm e as executa em seu cluster.

4.3.4 Flux

O Flux é um *framework* ou um conjunto de utilidades que permite facilitar a criação e operação de topologias. Originalmente, cada topologia em Storm é gerada utilizando um código Java. Cada vez que se necessita modificar uma topologia, é preciso recompilar e reempacotar o arquivo jar. Flux baseia-se na ideia de remover a lógica do arquivo jar, utilizando arquivos externos para definir e configurar suas topologias.

TEMA 5 – FLINK

Flink é um *framework* e um motor de processamento distribuído de dados limitados e ilimitados. Ele foi projetado para executar em todos os ambientes de *cluster*, realizar operações computacionais em velocidade de memória e em qualquer escala, além de permitir tanto o processamento de dados em lote quanto o processamento de fluxos de dados.

O motor de processamento de fluxos do Flink fornece alto desempenho e baixa latência. Aplicações Flink são tolerantes a falhas, ou seja, em caso de

algumas máquinas no *cluster* se tornarem indisponíveis, possivelmente por falha de rede ou de equipamento, as aplicações são capazes de manter o serviço operando normalmente. Além disso, o Flink é capaz de garantir o envio das mensagens. Programas para o Flink podem ser escritos em Java, Scala, Python ou SQL, e são compilados e recompilados automaticamente dentro do fluxo de dados no *cluster* em que são executados. Flink não oferece um sistema de armazenamento próprio. Porém fornece conectores para tais sistemas como: Amazon Kinesis, Apache Kafka, Alluxio, HDFS, Apache Cassandra e ElasticSearch.

5.1 Principais conceitos

Todos os dados em Flink são tratados como fluxos de eventos. Assim como muitos tipos de dados são gerados em fluxos, por exemplo: transações de cartão de crédito, *logs* de máquina, interações de usuário em um website ou aplicação *mobile*. Dessa forma os dados podem ser divididos em fluxos limitados, ou fluxos ilimitados.

- **Fluxos ilimitados:** possuem um começo, mas não possuem um fim definido. Não terminam e fornecem dados assim que eles são produzidos. Fluxos ilimitados devem ser processados continuamente, ou seja, eventos devem ser tratados tão logo sejam recebidos. Não é possível aguardar o recebimento de todos os dados, pois a entrada dos dados não termina em momento algum. Processar dados ilimitados normalmente exige que os dados sejam recebidos em uma ordem específica, assim como a ordem em que os eventos ocorrem, para ser capaz de avaliar a integridade dos resultados;
- **Fluxos limitados:** possuem um começo e um fim bem definidos. Fluxos limitados podem ser processados após a entrada de todos os dados antes da realização de qualquer computação. Os dados não exigem serem processados em uma ordem específica, uma vez que os dados podem ser ordenados a qualquer momento. O processamento de fluxos limitados também é conhecido como processamento em lote.

O controle preciso de tempo e estado permitem que o Flink execute qualquer tipo de aplicação em fluxos ilimitados. Fluxos limitados são processados internamente por algoritmos e estruturas de dados especificamente

projetadas para conjuntos de dados de tamanho fixo, o que permite que o processamento ocorra de forma otimizada.

O Flink é um sistema distribuído e exige recursos computacionais para executar aplicaçõesm sendo capaz de se integrar com qualquer gerenciador de recursos de *cluster* moderno, tais como: Hadoop YARN, Apache Mesos e Kubernetes, mas também pode ser executado de forma isolada. Flink é capaz de identificar automaticamente os recursos necessários baseado na configuração de paralelismo e os requisita aos gerenciadores de recursos. Em caso de falhas, o Flink substitui o *container* falho requisitando um novo recurso ao gerenciador de recursos. Toda a comunicação e requisição de recursos ocorre por meio de chamadas REST de forma a facilitar a integração do Flink com diversos ambientes.

O Flink é projetado para executar aplicações de fluxos de eventos com estados em qualquer escala. Aplicações são paralelizadas em milhares de tarefas que são distribuídas e executadas concorrentemente em um *cluster*. Portanto, uma aplicação pode escalar horizontalmente de forma virtualmente ilimitada. Além disso, o Flink é capaz de manter o estado de aplicações muito grandes. O Flink utiliza um algoritmo assíncrono e incremental de *checkpoints* garante que o impacto da latência computacional seja mínimo e ainda capaz de garantir a entrega de mensagens.

Aplicações que utilizam estados em Flink são otimizados para acessar estados locais. O estado de uma tarefa é mantido em memória, ou se o tamanho do estado excede a disponibilidade de memória da máquina, estruturas de dados são armazenadas em disco de forma eficiente. Por isso, tarefas realizam toda a computação acessando estados localmente em memória, no caso ideal. Flink garante a consistência dos estados em caso de falhas periodicamente e assincronamente armazenando os estados locais em forma de *checkpoints* utilizando um armazenamento durável.

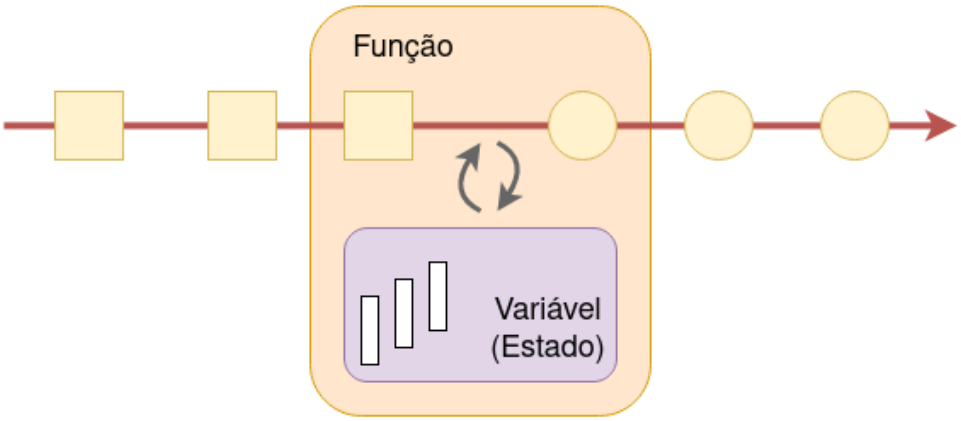
O Flink utiliza um conjunto de conceitos que são utilizados para definir como as aplicações são criadas e como executam. São eles:

- **Fluxos (*streams*):** fluxos podem ter diferentes características que afetam como um fluxo pode e deve ser processado. Os fluxos podem ser tanto limitados quanto ilimitados, como já vimos anteriormente. Quanto ao processamento, os fluxos podem ser tanto processados em tempo real quanto em lotes, ou seja, de forma gravada. Todos os dados são tratados

como fluxos, os quais podem ser processados em tempo real, assim que são gerados, ou de forma gravada, ou seja, os dados são recebidos integralmente para então serem processados;

- **Estado (*state*):** cada aplicação de fluxo não trivial possui estados, ou seja, apenas aplicações que aplicam transformações em eventos individuais não possuem estados. Qualquer aplicação que execute regras básicas de negócio precisa guardar os eventos ou resultados intermediários para acessá-los em outro momento, por exemplo, quando o próximo evento for recebido ou após um tempo específico. O Flink fornece estados primitivos para diferentes estruturas de dados, tais como valores atômicos, listas ou mapas. Os estados de uma aplicação são gerenciados por processos plugáveis de estados. Flink oferece diversos processos de estados que armazenam estados em memória, no RockDB, uma forma de armazenamento em disco incorporado nativamente, ou outro processo customizado de armazenamento. Os *checkpoints* e algoritmos de recuperação garantem a consistência dos estados da aplicação em caso de falha. Por isso falhas ocorrem de forma transparente e não afetam a correta operação da aplicação. Flink é capaz de manter os estados de vários *terabytes* de tamanho devido ao algoritmo de *checkpoint* incremental e assíncrono.

Figura 11 – Estado em um fluxo em Flink



Fonte: Flink, 2019.

- **Tempo:** tempo é outro conceito importante em aplicações de fluxos. A maioria dos fluxos de eventos possuem uma semântica temporal inerente pois cada evento é criado em um ponto específico no tempo. Além disso,

várias operações de fluxo são dependentes de tempo, tais como agregações de janelas, sessões, detecção de padrões, e *joins* baseados em tempo. Um importante aspecto de processamento de fluxos é como uma aplicação mede o tempo, ou seja, a diferença entre o tempo do evento e o tempo do processamento.

5.2 APIs em camadas

O Flink oferece um conjunto de APIs em três camadas. Cada uma das camadas fornece uma troca entre concisão e expressividade e cada um deles é mais adequado para diferentes casos de uso.

5.2.1 *Process Functions*

Process Functions é a interface mais complexa que o Flink oferece. Tal interface é adequada para processar um ou dois fluxos de entrada ou eventos que foram agrupados em janelas. *Process Functions* oferecem um controle preciso sobre tempo e estados. Um *Process Function* pode modificar arbitrariamente seus estados e registrar temporizadores que chamarão funções de *callback* em um tempo futuro. *Process Funtions* podem implementar uma lógica de negociação por evento tão complexa for necessária pela aplicação.

5.2.2 Data Stream API

A *Data Stream* API fornece um conjunto de primitivas muito comuns para muitas operações de processamento de fluxo. Essas operações incluem: janelas, gravação por tempo, transformações e enriquecimento ou melhoramento de eventos por meio de consultas em sistemas de armazenamento de dados externos. Esta API está disponível para as linguagens Java e Scala e é baseada nas funções *map(), reduce()* e *aggregate().* Funções podem ser definidas ao se *extender* interfaces ou por meio de funções lambda em Java ou Scala.

5.2.3 SQL e Table API

O Flink oferece duas API relacionais: a SQL e a Table API. Ambas as APIs são utilizadas tanto para processamento de fluxos quanto em lote, ou seja, as consultas são executadas com a mesma semântica em fluxos limitados ou

ilimitados e produzindo o mesmo tipo de resultados. As APIs utilizam o Apache Calcite para análise, validação e otimização de consultas.

FINALIZANDO

Nesta aula nos aprofundamos nos temas relacionados ao processamento de fluxos, e as principais aplicações utilizadas para ingestão de dados e processamento de fluxos.

Iniciamos com uma breve explicação sobre os conceitos mais importantes no processamento de fluxos distribuídos e uma explicação mais aprofundada da arquitetura lambda.

Em seguida, a primeira plataforma de processamento de fluxos que conhecemos é a Kafka. Exploramos seus principais conceitos e sua utilização com como um sistema de mensagem, um sistema de armazenamento e um sistema de processamento de fluxos. Também conhecemos o sistema distribuído para processamento de fluxos Flume. Exploramos seus conceitos, arquitetura e funcionamento. Além disso, verificamos um pequeno exemplo prático. Na sequência, conhecemos o sistema distribuído de processamento de fluxos Storm. Exploramos seus principais conceitos, como topologias, *spouts* e *bolts*, sua arquitetura e algumas interfaces para a construção de topologias. Por fim, conhecemos o *framework* e motor de processamento de fluxos Flink. Exploramos seus principais conceitos e suas APIs utilizadas para implementar a lógica de processamento dos fluxos.

REFERÊNCIAS

APACHE FLUME. **Flume 1.9.0 User Guide**. Wilmington, DE: The Apache Software Foundation, 2019. Disponível em: <https://flume.apache.org/releases/content/1.9.0/FlumeUserGuide.html>. Acesso em: 24 nov. 2020.

APACHE SOFTWARE FOUNDATION. Apache Storm – Tutorial. Wilmington, DE: The Apache Software Foundation, 2019.

FLINK. What is Apache Flink? – Applications. **Flink**, 2019. Disponível em: <https://flink.apache.org/flink-applications.html>. Acesso em: 24 nov. 2020.

KAFKA APACHE. Documentation. **Kafka Apache**, S.d. Disponível em: <https://kafka.apache.org/documentation/>. Acesso em: 24 nov. 2020.

MARZ, N; WARREN, J. **Big Data:** principles and best practices of scalable realtime data systems. New York, NY: Manning, 2015.